2

# WORKING MATERIAL

FOR THE LECTURES OF

GERARD HUET

# A UNIFORM APPROACH TO TYPE THEORY

INTERNATIONAL SUMMER SCHOOL

ON

## LOGIC, ALGEBRA AND COMPUTATION

MARKTOBERDORF, GERMANY, JULY 25 - AUGUST 6,1989

THIS SUMMER SCHOOL IS ORGANIZED UNDER THE AUSPICES OF THE TECHNISCHE UNIVERSITÄT MÜNCHEN AND IS SPONSORED BY THE NATO SCIENCE COMMITEE AS PART OF THE 1989 ADVANCED STUDY INSTITUTES PROGRAMME, PARTIAL SUPPORT FOR THE CONFERENCE WAS PROVIDED BY THE EUROPEAN RESEARCH OFFICE AND THE NATIONAL SCIENCE FOUNDATION AND BY VARIOUS INDUSTRIAL COMPANIES.

# A Uniform Approach to Type Theory

Gérard Huet

INRIA

The author presents

## Abstract

~~We present~~ in a unified framework the basic syntactic structures used to model deductive and computational notions. The guiding principle is "propositions as types".

These course notes have been prepared for the Institute on Logical Foundations of Fonctional Programming organized by the University of Texas at Austin in June 1987. They will appear in "Logical Foundations of Fonctional Programming", Addison-Wesley, 1989.

# 1 Terms and types

## 1.1 General notations

We assume known elementary set theory and algebra. $\mathcal{N}$ is the set $\{0, 1, ...\}$ of natural numbers, $\mathcal{N}_+$ the set of positive natural numbers. We shall identify the natural $n$ with the set $\{0, ..., n-1\}$, and thus 0 is also the empty set $\emptyset$. Every finite set $S$ is isomorphic to $n$, with $n$ the cardinal of $S$, denoted $n = |S|$. If $A$ and $B$ are sets, we write $A \rightarrow B$, or sometimes $B^A$, for the set of functions with domain $A$ and codomain $B$.

## 1.2 Languages, concrete syntax

Let $\Sigma$ be a finite alphabet. A *string* $u$ of *length* $n$ is a function in $n \rightarrow \Sigma$. The set of all strings over $\Sigma$ is

$$\Sigma^* = \bigcup_{n \in \mathcal{N}} \Sigma^n.$$

We write $|u|$ for the length $n$ of $u$. We write $u_i$ for $u(i-1)$, when $i \leq n$. The null string, unique element of $\Sigma^0$, is denoted $\Lambda$. The unit string mapping 1 to $a \in \Sigma$ is denoted "$a$". The concatenation of strings $u$ and $v$, defined in the usual fashion, is denoted $u \mathbin{\hat{}} v$, and when there is no ambiguity we write e.g. "$abc$" for "$a$" $\hat{}$ "$b$" $\hat{}$ "$c$". When $u \in \Sigma^*$ and $a \in \Sigma$, we write $u \cdot a$ for $u \mathbin{\hat{}}$ "$a$". We define an ordering $\leq$ on $\Sigma^*$, called *the prefix ordering*, by

$$u \leq v \Leftrightarrow \exists w \quad v = u \mathbin{\hat{}} w.$$

If $u \leq v$, the residual $w$ is unique, and we write $w = v/u$. We say that strings $u$ and $v$ are *disjoint*, and we write $u|v$, iff $u$ and $v$ are unrelated by the partial ordering $\leq$. Finally we let $u < v$ iff $u \leq v$ with $u \neq v$.

The set $\Sigma^*$ has the structure of a monoïd, that is:

$$Ass: (u \mathbin{\hat{}} v) \mathbin{\hat{}} w = u \mathbin{\hat{}} (v \mathbin{\hat{}} w)$$

$$IdL: \Lambda \mathbin{\hat{}} u = u$$

$$IdR: u \mathbin{\hat{}} \Lambda = u.$$

Actually, $\Sigma^*$ is the free monoïd generated by $\Sigma$.

**Examples.**

1. $\Sigma = 0$. We get $\Sigma^* = 1$.
2. $\Sigma = 1$. We get $\Sigma^* = \mathcal{N}$. Strings are here natural numbers in unary notation, and concatenation corresponds to addition.
3. $\Sigma = 2 = \{0, 1\}$ (the Booleans). The set $\Sigma^*$ is the set of all binary words.
4. $\Sigma = \mathcal{N}_+$. We call the elements of $\Sigma^*$ *occurrences*. When $u = w \cdot m$ and $v = w \cdot n$, with $m < n$, we say that $u$ is *left* of $v$, and write $u <_L v$.

### 1.3 Terms: abstract syntax

We first define a *tree domain* as a subset $D$ of $\mathcal{N}_+^*$ closed under $<$ and $<_L$:

$$u \in D \wedge v < u \Rightarrow v \in D$$

$$u \in D \wedge v <_L u \Rightarrow v \in D.$$

We say that $M$ is a $\Sigma$-tree iff $M \in D \to \Sigma$, for some tree domain $D$. We define $D(M)$ as $D$, and we say that $D(M)$ is the *set of occurrences* in $M$. $M$ is said to be *finite* whenever $D(M)$ is, which we shall assume in the following.

We shall now use occurrences to designate nodes of a tree, and the subtree starting at that node. If $u \in D(M)$, we define the $\Sigma$-tree $M/u$ as mapping occurrence $v$ to $M(u \mathbin{\hat{}} v)$. We say that $M/u$ is the sl *subtree of* $M$ at occurrence $u$. If $N$ is also a $\Sigma$-tree, we define the *graft* $M[u \leftarrow N]$ as the $\Sigma$-tree mapping $v$ to $N(w)$ whenever $v = u \mathbin{\hat{}} w$ with $w \in D(N)$, and to $M(v)$ if $v \in D(M)$ and not $u \leq v$.

We need one auxiliary notion, that of *width* of a tree. If $M \in \Sigma^*$, we define the (top) width of $M$ as

$$||M|| = \max\{n \mid \text{“}n\text{”} \in D(M)\}.$$

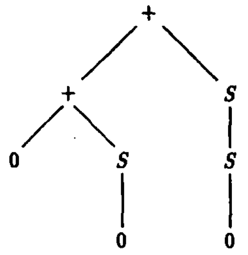
We shall now consider $\Sigma$ a *graded* alphabet, that is given with an *arity* function $\alpha$ in $\Sigma \to \mathcal{N}$. We then say that $M$ is a $\Sigma$-*term* iff $M$ is a $\Sigma$-tree verifying the supplementary consistency condition:

$$\forall u \in D(M) \ \ ||M/u|| = \alpha(M(u)).$$

That is, every subtree of $M$ is of the form $F(M_1, M_2, ..., M_n)$, with $n = \alpha(F)$. We write $T(\Sigma)$ for the set of $\Sigma$-terms. If $M_1, M_2, ...M_n \in T(\Sigma)$ and $F \in \Sigma$, with $\alpha(F) = n$, then $M = F(M_1, M_2, ...M_n)$ is easily defined as a $\Sigma$-term. This gives $T(\Sigma)$ the structure of a $\Sigma$-algebra. Since conversely the decomposition of $M$ is uniquely determined, we call $T(\Sigma)$ the *completely free* $\Sigma$-algebra.

**Example**

With $\Sigma = \{+, S, 0\}$, $\alpha(+) = 2$, $\alpha(S) = 1$, $\alpha(0) = 0$, the following structure represents a $\Sigma$-term:



The following proposition is easy to prove by induction. All occurrences are supposed to be universally quantified in the relevant tree domain.

**Proposition 1.**

$$\begin{aligned}
\textit{Embedding}: \quad & M[u \leftarrow N]/(u \mathbin{\hat{}} v) = N/v \\
\textit{Associativity}: \quad & M[u \leftarrow N][u \mathbin{\hat{}} v \leftarrow P] = M[u \leftarrow N[v \leftarrow P]] \\
\textit{Persistence}: \quad & M[u \leftarrow N]/v = M/v \quad (u|v) \\
\textit{Commutativity}: \quad & M[u \leftarrow N][v \leftarrow P] = M[v \leftarrow P][u \leftarrow N] \quad (u|v) \\
\textit{Distributivity}: \quad & M[u \leftarrow N]/v = (M/v)[u/v \leftarrow N] \quad (v \leq u) \\
\textit{Dominance}: \quad & M[u \leftarrow N][v \leftarrow P] = M[v \leftarrow P] \quad (v \leq u).
\end{aligned}$$

We define the *length* $|M|$ of a (finite) term $M$ recursively by:

$$|F(M_1, ..., M_2)| = 1 + \Sigma_{i=1}^{n} |M_i|.$$

### 1.4 Parsing

It is well-known that the term in the example above can be represented unambiguously as a $\Sigma$-string, for instance in *prefix polish notatiou*, that is here: $++0S0SS0$. This result is not very interesting: such strings are neither good notations for humans, nor good representations for computers, since the graft operation necessitates unnecessary copying. We shall discuss later better machine representations, using binary graphs. As far as human readibility is concerned, we assume known parsing techniques. This permits to represent terms, on an extended alphabet with parentheses and commas, which is closer to standard mathematical practice. Also, infix notation and indentation permit to keep in the string some of the tree structure more apparent. We shall not make explicit the exact representation grammar, and allow ourselves to write freely for instance $(0 + S(0)) + S(S(0))$. Note that we avoid explicit quotes as well, which permits us to mix freely meta-variables with object structures, like in $S(M)$, where $M$ is a meta-variable denoting a $\Sigma$-term.

### 1.5 Terms with variables, substitution

The idea is to internalize the notation $S(M)$ above as a term $S(x)$ over an extended alphabet containing special symbols of arity 0 called *variables*. Such terms with variables are thus polynomial expressions, in the case of completely free operators.

Let $V$ be a denumerable set disjoint from $\Sigma$. We define the set of terms with variables, $T(\Sigma, V)$, in exactly the same way as $T(\Sigma \cup V)$, extending the arity function so that $\alpha(x) = 0$ for every $x$ in $V$. The only difference between the variables and the constants (symbol of arity 0) is that a constant has an existential import: it denotes a value in the domain we are modelling with our term language, whereas a variable denotes a term. The difference is important only when there are no constants in $\Sigma$, since then $T(\Sigma)$ is empty.

All of the notions defined for terms extend to terms with variables. We define the set $V(M)$ of *variables* occurring in $M$ as:

$$V(M) \;=\; \{x \in V \mid \exists u \in D(M) \quad M(u) = x\},$$

and we define the number of distinct variables in $M$ as $\nu(M) \;=\; |V(M)|$.

We shall now formalize the notion of substitution of terms for variables in a term containing variables. From now on, the sets $\Sigma$ and $V$ are fixed, and we use $T$ to denote $T(\Sigma, V)$. A *substitution* $\sigma$ is a function in $V \to T$, identity almost everywhere. That is, the set $D(\sigma) \;=\; \{x \in V \mid \sigma(x) \neq x\}$ is finite. We call it the *domain* of $\sigma$. Substitutions are extended to $\Sigma$-morphisms over $T$ by

$$\sigma(F(M_1, ..., M_n)) \;=\; F(\sigma(M_1), ..., \sigma(M_n)).$$

Bijective substitutions are called *permutations*. When $U \subseteq V$, we write $\sigma_U$ for the restriction of substitution $\sigma$ to $U$. It is easy to show that, for all $\sigma$, $M$ and $U$:

$$V(M) \subseteq U \;\Rightarrow\; \sigma(M) = \sigma_U(M).$$

Alternatively, we can define the replacement $M[x \leftarrow N]$ as

$$M[u_1 \leftarrow N] ... [u_n \leftarrow N],$$

where $\{u_1, ..., u_n\} \;=\; \{u \mid M(u) = x\}$ and then

$$\sigma(M) \;=\; M[x \leftarrow \sigma(x) \mid x \in V(M)]$$

with an obvious notation.

We now define the quasi-ordering $\leq$ of *matching* in $T$ by:

$$M \leq N \;\Leftrightarrow\; \exists \sigma \quad N = \sigma(M).$$

It is easy to show that if such a $\sigma$ exists, $\sigma_{V(M)}$ is unique. We shall call it the match of $N$ by $M$, and denote it by $N/M$.

We define $M \equiv N \;\Leftrightarrow\; M \leq N \;\wedge\; N \leq M$. When $M \equiv N$, we say that $M$ and $N$ are *isomorphic*. This is equivalent to say that $M = \sigma(N)$ for some permutation $\sigma$. Note that $M \equiv N$ implies $|M| = |N|$. Finally, we define

$$M > N \;\Leftrightarrow\; N \leq M \;\wedge\; \neg M \leq N.$$

**Proposition.** $>$ is a well-ordering on $T$.

Proof. We show that $M > N$ implies $\mu(M) > \mu(N)$, with $\mu(M) \;=\; |M| - \nu(M)$.

Let $\varphi$ be any bijection between $T \times T$ and $V$. We define a binary operation $\sqcap$ in $T$ by:

$$F(M_1, ..., M_n) \sqcap F(N_1, ..., N_n) \;=\; F(M_1 \sqcap N_1, ..., M_n \sqcap N_n)$$

$$M \cap N = \varphi(M,N) \quad \text{in all other cases.}$$

$M \cap N$ is uniquely determined from $\varphi$ and, for distinct $\varphi$'s, is unique up to $\equiv$.

**Proposition.** $M \cap N$ is a g.l.b. of $M$ and $N$ under the match quasi-ordering.

Let **T** be the quotient poset $T/\equiv$, completed with a maximum element $\top$. From the propositions above we conclude:

**Theorem.** **T** is a complete lattice.

**Corollary.** If two terms $M$ and $N$ have an upper bound, i.e. a common instance $\sigma(M) = \sigma'(N)$, they have a l.u.b. $M \cup N$, which is a most general such instance; that is, $\sigma = \sigma_0 \circ \tau$, and $\sigma' = \sigma_0' \circ \tau$, for some substitution $\tau$ called the *principal unifier* of $M$ and $N$. The term $M \cup N$ is unique modulo $\equiv$ and may be found by the *unification* algorithm [159].

**Proposition.**

$$D(\sigma(M)) = D(M) \cup \bigcup_{\{u \mid M(u) \in V\}} \{u \mathbin{^\frown} v \mid v \in D(\sigma(M(u)))\}$$

$$\forall u \in D(M) \;\; M(u) = x \in V \Rightarrow \forall v \in D(\sigma(x)) \; \sigma(M)/(u \mathbin{^\frown} v) = \sigma(x)/v$$

$$\forall u \in D(M) \;\; \sigma(M)/u = \sigma(M/u)$$

$$\forall u \in D(M) \;\; \sigma(M)[u \leftarrow \sigma(N)] = \sigma(M[u \leftarrow N]).$$

## 1.6 Graph representations, dags

It is usual to represent trees in computers by binary graphs implemented as pairs of machine words. In the simplest scheme, a word is partitioned into one tag bit, and one field interpreted either as an address in the graph memory, or as a natural number, according to the value of the tag. In this last case, some natural (say 0) is reserved for *nil*, the empty list of trees. Symbols from $\Sigma$ are then coded up as positive naturals. If tree $M$ is represented by the word $W$ and the list $L$ is represented by the word $W'$, then the list $M \cdot L$ is represented by the address of a graph node implemented as the pair $(W, W')$. Similarly, if symbol $F$ is coded up as the word $W$ and the list $L$ is represented by the word $W'$, then the tree $F(L)$ is represented by the address of a graph node implemented as the pair $(W, W')$.

Thus every tree is mapped into a graph, and this representation allows sharing of common subtrees. Assignment to fields may implement grafting without copying, but this method is not usually compatible with sharing. This is the standard way of representing trees and lists in symbol-manipulation languages such as LISP [124]. The principal problem to be solved in such languages is to keep track dynamically of which areas of the storage are used to represent actively used subtrees. Garbage-collection algorithms have been proposed to solve this problem, but this method is becoming problematic with the current technology of very large virtual memories. A precise description of such memory allocation issues is beyond the scope of these notes.

Terms are of course represented as trees. A global table holds the arity function. There are several possibilities for the representation of variables. They may be represented as symbols. But then the scope structure must be computed by an algorithm, rather than being implicit in the structure. Also a global scanning of the term is necessary to determine its set of variables, and substitution involves copying of the substituted term. For these reasons, variables are often represented rather as integer offsets in stacks of bindings. Such "structure sharing" representations are now standard for PROLOG implementations.

A precise account of the various representations schemes for term structures, and of the accompanying algorithms, is out of the scope of these notes. It should be born in mind that the crucial problem is memory utilization: the trade-off between copying and sharing is often the deciding factor for an *implementation*. Languages with garbage-collected structures, such as LISP, are ideal for programming "quick and dirty" prototypes. But serious implementation efforts should aim at good algorithmic performance on realistic size applications.

The crucial algorithms in formula and proof manipulation are matching, unification, substitution and grafting. First-order unification has been specially well studied. A linear algorithm is known [142,28], but in practice quasi-linear algorithms based on congruence classes operations are preferred [115,116]. Furthermore, these algorithms extend without modification to unification of infinite rational terms represented by finite graphs [77].

Implementation methods may be partitioned into two families. Some depend on logical properties (e.g. sharing subterms in dags arising from substitution to a term containing several occurrences of the same variable). Some are purely statistical (e.g. sharing structures globally through hash-coding techniques). Particular applications require a careful analysis of the optimal trade-off between logical and statistical techniques.

There is no comprehensive survey on implementation issues. Some partial aspects are described in [9,160,117,115,189,184,135,52,1,43,54,20,57,168,185].

## 2 Inference rules

We shall now study *inference systems*, defined by inference rules. The general form of an inference rule is:

$$IR : \frac{P_1\ P_2\ \ldots\ P_n}{Q},$$

where the $P_i$'s and $Q$ are *propositions* belonging to some formal language. We shall here regard these propositions as *types*, and the inference rule as the description of the signature of $IR$ considered as a typed operator. More precisely, $IR$ has arity $n$, $P_i$ is the type of its $i$-th argument, and $Q$ is the type of its result. Well-typed terms composed of inference operators are called the *proofs* defined by the inference system. Let us now examine a few familiar inference systems.

### 2.1 The trivial homogeneous case: Arities

A graded alphabet $\Sigma$ may be considered as the simplest inference systems, where types are reduced to arities. I.e., the set of propositions is 1, and an operator $F$ of arity $n$ is an inference rule

$$F : \frac{0\ 0 \ldots 0}{0}$$

(with $n$ zero's in the numerator). A $\Sigma$-proof corresponds to our $\Sigma$-terms above.

### 2.2 Finite systems of types: Sorts

The next level of inference systems consist in choosing a finite set $S$ of elementary propositions, usually called *sorts*. For instance, let $S = \{int, bool\}$, and $\Sigma$ be defined by:

$$0 : int \qquad S : int \rightarrow int \qquad true : bool \qquad false : bool \qquad if : bool, int, int \rightarrow int,$$

where we use the alternative syntax $P_1, \ldots, P_n \rightarrow Q$ for an inference rule. The term $if(true, 0, S(0))$ is of sort $int$, i.e. it is a proof of proposition $int$.

As another example, consider the puzzle "Missionaries and Cannibals". We call *configuration* any triple $\langle b, m, c\rangle \in 2 \times 4 \times 4$. The boolean $b$ indicates the position of the boat, $m$ (resp. $c$) is the number of missionaries (resp. cannibals) on the left bank. The set of states $S$ is the set of *legal* configurations, that obey the condition

$$P(m,c) \;\equiv\; m = c \text{ or } m = 0 \text{ or } m = 3.$$

There are thus 10 distinct states or sorts. The rules of inference comprise first a constant denoting the starting configuration:

$$s_0 : \langle 0,3,3\rangle$$

then the transitions carrying $p$ missionaries and $q$ cannibals from left to right:

$$L_{m,c,p,q} : \langle 0,m,c\rangle \to \langle 1, m-p, c-q\rangle \quad (m \geq p, c \geq q, P(m,c), P(m-p,c-q), 1 \leq p+q \leq 2)$$

and finally the transitions $R_{m,c,p,q}$, which are inverses of $L_{m,c,p,q}$. The game consists in finding a proof of $\langle 1,0,0\rangle$.

This simple example of a finite group of transformations applies to more complex tasks, such as Rubik's cube. All state transition systems can be described in a similar fashion. Examples of such proofs are parse-trees of regular grammars, where the inference rules signatures correspond to a finite automaton transition graph. Slightly more complicated formalisms allow subsorts, i.e. containment relationships between the sorts. That is, we postulate primitive implications between the elementary propositions. These systems reduce to simple sorts by considering dummy transitions corresponding to the implicit coercions.

### 2.3 Types as terms: standard proof trees

We shall here describe our types as terms formed over an alphabet $\Phi$ of type operators, which we shall call *functors*. For the moment, we shall assume that we have just one category of such propositions, i.e. the functors have just an arity. The alphabet $\Sigma$ of inference rules determines the legal proof trees.

**Example: Combinatory logic.**

We take as functors a set $\Phi$ of constants $\Phi_0$, plus a binary operator $\Rightarrow$, which we shall write in infix notation. We call *functionality* a term in $T(\Phi)$. We have three families of rules in $\Sigma$. In the following, the meta-variables $A, B, C$ denote arbitrary functionalities. The operators of the $K$ and $S$ families are of arity 0, the operators of the *App* family are binary.

$$K_{A,B} : A \Rightarrow (B \Rightarrow A)$$

$$S_{A,B,C} : (A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$$

$$App_{A,B} : \frac{A \Rightarrow B \quad A}{B}.$$

Here is an example of a proof. Let $A$ and $B$ be any functionalities, $C = B \Rightarrow A$, $D = A \Rightarrow C$, $E = A \Rightarrow A$, $F = A \Rightarrow (C \Rightarrow A)$, $G = D \Rightarrow E$. The term

$$App_{D,E}(App_{F,G}(S_{A,C,A}, K_{A,C}), K_{A,B})$$

has type $E$, i.e. it gives a proof of the proposition $A \Rightarrow A$.

We express formally that proof $M$ proves proposition $P$ in the inference system $\Sigma$ as:

$$\Sigma \vdash M : P$$

. That is, we think of a theorem as the type of its proof tree. Proof-checking is identified with type-checking. Here this is a simple consistency check; that is, if operator $F$ is declared in $\Sigma$ as: $F : P_1, ..., P_n \rightarrow Q$ and if $\Sigma \vdash M_i : P_i$ for $1 \leq i \leq n$, then $\Sigma \vdash F(M_1, ..., M_n) : Q$.

### 2.4 Polymorphism: Rule schemas

This next level of generality consists in allowing variables in the propositional terms. This is very natural, since it internalizes the meta-variables used to index families of inference rules as propositional variables. The rules of inference become thus *polymorphic* operators, whose types are expressions containing free variables. This is the traditional notion of schematic inference rule from mathematical logic: each rule is a schema, denoting a family of operators, whose types are all instances of the clause.

**Example.** The example from the previous section is more naturally expressed in this polymorphic formalism. We replace the set $\Phi_0$ by a set of variables $V$, and now we have just 3 rules of inference: $K$, $S$ and $App$.

Type-checking is now explained in terms of instantiation. Let $\Sigma$ be the current signature of polymorphic operators. We define what *it means for a tree $T$ to be consistently typed* of type $\tau$ in theory $\Sigma$, which we write $\Sigma \vdash T : \tau$. The definition is by induction on the size of $T$. Assume that $F : Q_1; Q_2; ...Q_n \rightarrow P$ is in $\Sigma$, and that for some substitution $\sigma$ we have $\Sigma \vdash T_i : \sigma(Q_i)$ for all $1 \leq i \leq n$. Then we get $\Sigma \vdash F(T_1, ..., T_n) : \sigma(P)$.

The types can actually be completely dispensed with, since a well typed term possesses a most general type, called its *principal type*. For instance, in the example above, the proof $App(App(S, K), K)$ has the principal type $A \Rightarrow A$, with $A \in V$. This term is usually written $I = SKK$ in combinatory logic, where the concrete syntax convention is to write combinator strings to represent sequences of applications associated to the left.

The notion of principal type, first discovered by Hindley in the combinatory logic context, and independently by Milner foı ML type-checking [129], is actually completely general:

**The Principal Type Theorem.** Let $\Sigma$ be any signature of polymorphic operators over a functor signature $\Phi$. Let $M$ be a legal proof term. Then $M$ possesses a principal type $\tau \in T(\Phi, V)$. That is, $\Sigma \vdash M : \tau$, and for all $\tau' \in T(\Phi, V)$, $\Sigma \vdash M : \tau'$ implies $\tau \leq \tau'$.

Proof. Simple induction, using the properties of the principal unifier. Let $T = F(T_1, ..., T_n)$, with $\Sigma \vdash T : M$. This means that $F : Q_1; Q_2; ...Q_n \rightarrow P$ is in $\Sigma$, and that $M = \sigma(P)$, with $\Sigma \vdash T_i : \sigma(Q_i)$. By the induction hypothesis, $\Sigma \vdash T_i : \tau_i$, with $\tau_i$ principal. Thus for some $\rho_i$ we have $\sigma(Q_i) = \rho_i(\tau_i)$. We may assume without loss of generality that the $\tau_i$ are renamed so that they have no variable in common, and no variable in common with the defining clause for $F$. Thus the tuples $< ..., Q_i, ... >$ and $< ..., \tau_i, ... >$ are simultaneously unifiable, and their principal unifier $\theta$ gives a tuple $< ..., N_i, ... >$ such that $N_i = \theta(Q_i) = \theta(\tau_i)$. The construction defines $\tau = \theta(P)$ having the required properties.

By now we have developed enough formalism to make sense out of our "propositions as types" paradigm. Actually, the example we have discussed above is the fragment of propositional logic known as "minimal logic". When regarding the functor $\Rightarrow$ as (intuitionistic) implication, and $App$

as the usual inference rule of Modus ponens, $K$ and $S$ are the two axioms of minimal logic presented as a Hilbert calculus. Combinatory logic is thus the calculus of proofs in minimal logic [48,103].

Actually combinators don't just have a type, they have a value. They can be *defined* with definition equations in terms of application. Using the concrete syntax mentioned above, we get for instance $K$ and $S$ defined by the following equations:

$$Def_K : \; K\ x\ y \; = \; x$$

$$Def_S : \; S\ x\ y\ z \; = \; x\ z\ (y\ z).$$

**Exercise.** Verify that the two equations above, when seen as unification constraints, define the expected principal types for $K$ and $S$.

This point of view of considering equality axiomatizations of the proof structures corresponds to what the proof-theorists call *cut elimination*. That is, the two equations above can be used as rewrite rules in order to eliminate redundancies corresponding to useless detours in the proofs. We shall develop more completely this point of view of *computation as proof normalization* below.

The current formalism of inference rules typed by terms with variables corresponds to intuitionistic sequents in proof theory, and to Horn clauses in automated reasoning. For instance, a PROLOG [31] interpreter may be seen in this framework as a proof synthesis method. Given an alphabet $\Sigma$ of polymorphic inference rules (usually called definite clauses), and a proposition $\tau$ over functor alphabet $\Phi$, it returns (when possible) a proof term $M$ such that $M$ is a legal $\Sigma$-proof term consistently typed with type $\tau'$ instance of $\tau$:

$$\Sigma \vdash M : \tau' \geq \tau.$$

With $\sigma$ the principal unifier of $\tau$ and the principal type of $M$, we say that $\sigma$ is a PROLOG *answer* to the *query* $\tau$. Of course this explanation is incomplete; we have to explain that PROLOG finds all such instances by a backtrack procedure constructing proofs in a bottom-up left-to-right fashion, using operators from $\Sigma$ in a specific order (the order in which clauses are declared); this last requirement leads to incompleteness, since PROLOG may loop with recursively composable operators, whereas a different order might lead to termination of the procedure. Also, PROLOG may be presented several goals together, and they may share certain variables, but this may be explained by a simple extension of the above proof-synthesis explanation.

We claim that this explanation of PROLOG is more faithful to reality than the usual one with Horn clauses. In particular, our explanation is completely constructive, and we do not have to explain the processes of conjunctive normalization and Skolemization. Furthermore, there is no distinction in $\Phi$ between predicate and function symbols, consistently with most PROLOG implementations. Actually, we even allow polymorphic signatures which would not be accepted as definite clauses, since some of the types may be reduced to single variables, like for *App* above.

## 2.5 Allowing lemmas

The next convenience in a general formalism for manipulating proofs consists in providing the user with a facility to derive and use lemmas.

Let us use the notation $\pi_\Sigma(M)$ to denote the principal type of the (legal) $\Sigma$-term $M$. Thus we have

$$\Sigma \vdash M : \pi_\Sigma(M).$$

It is now possible to use $M$ as a lemma, choosing to name it with a symbol *name* not in $\Sigma$, using the new term constructor: *let name* $=$ $M$ *in* $N$. The term $N$ is a proof term constructed in $\Sigma$, enriched with the new (nullary operator) constant *name*. More precisely, the legal proofs using lemmas are defined using the rule:

$$\Sigma \vdash \textit{let name} \;=\; M \textit{ in } N : \tau \Longleftrightarrow \Sigma \cup \{\textit{name} :\ \pi_\Sigma(M)\} \vdash N : \tau.$$

**Example.** Using the minimal logic combinators above, i.e. $\Sigma = \{K, S, App\}$, derive:

$$\Sigma \vdash \textit{let } I \;=\; S\ K\ K \textit{ in } I\ I \;\; : A \Rightarrow A.$$

This shows that constant $I$ is used in a polymorphic way, similarly to the basic combinators from $\Sigma$, since its two occurrences in $App(I, I)$ above are typed with two distinct instances of its principal type $\pi_\Sigma(S\ K\ K) \;=\; (A \Rightarrow A)$.

**Remark.** We might more generally expect a facility to use derived inference rules. But here we have a notational difficulty, in order to explain how the free variables from the principal types of the argument proofs are shared, since in the *let* constant declaration mechanism above we kept the type of $M$ implicit. In standard mathematical practice we think of lemmas as names of propositions (i.e. types) rather than proofs. Thus instead of derived inference rules we tend to use rather proofs modulo hypotheses. This level of term description corresponds to $\lambda$-calculus, which we shall now study.

## 3 Combinatory Algebra and $\lambda$-calculus

### 3.1 Proofs with variables: sequents

We first come back to the general theory of proof structures. We assume the alphabet $\Sigma$ of rules of inference to be fixed, and thus we abbreviate $\Sigma \vdash M : N$ as $\vdash_\Sigma M : N$ or even $\vdash M : N$ when $\Sigma$ is clear from the context.

We saw earlier that the Hilbert presentation of minimal logic was not very natural, in that the trivial theorem $A \Rightarrow A$ necessitated a complex proof $S\ K\ K$. The problem is that in practice one does not use just proof terms, but *deductions* of the form

$$\Gamma \vdash A$$

where $\Gamma$ is a set of (hypothetic) propositions.

Deductions are exactly proof terms *with variables*. Naming these hypothesis variables and the proof term, we write:

$$\{...[x_i : A_i]... \mid i \leq n\} \vdash M : A$$

with $V(M) \subseteq \{x_1, ..., x_n\}$. Such formulas are called *sequents*. Since this point of view is not very well-known, let us emphasize this observation:

**Sequents represent proof terms with variables.**

Note that so far our notion of proof construction has not changed:

$\Gamma \vdash_\Sigma M : A$ iff $\vdash_{\Sigma \cup \Gamma} M : A$, i.e. the hypotheses from $\Gamma$ are used as supplementary axioms, in the same way that in the very beginning we have defined $T(\Sigma, V)$ as $T(\Sigma \cup V)$.

In the next section, we assume fixed the combinatory algebra proof system: $\Sigma \;=\; \{K, S, App\}$.

### 3.2 The deduction theorem

This theorem, fundamental for doing proofs in practice, gives an equivalence between proof terms with variables and functional proof terms:

$$\Gamma \cup \{A\} \vdash B \;\Leftrightarrow\; \Gamma \vdash A \Rightarrow B$$

That is, in our notations:
a) $\Gamma \vdash M : A \Rightarrow B \;\Rightarrow\; \Gamma \cup \{x : A\} \vdash (M\ x) : B$
This direction is immediate, using App, i.e. Modus Ponens.
b) $\Gamma \cup \{x : A\} \vdash M : B \;\Rightarrow\; \Gamma \vdash [x]M : A \Rightarrow B$
where the term $[x]M$ is given by the following algorithm.

**Schönfinkel's abstraction algorithm:**

$$\begin{array}{lll} [x]x = I & (= S\ K\ K) \\ [x]M = K\ M & \text{if } M \text{ atom (variable or constant)} \neq x \\ [x](M\ N) = S\ [x]M\ [x]N \end{array}$$

Note that this algorithm motivates the choice of combinators $S$ and $K$ (and optionally $I$). Again we stress a basic observation:

**Schönfinkel's algorithm is the essence of the proof of the deduction theorem.**

Now let us consider the rewriting system $R$ defined by the rules:

$$Def_K : K\ x\ y = x,$$

$$Def_S : S\ x\ y\ z = ((x\ z)\ (y\ z)),$$

optionally supplemented by:

$$Def_I : I\ x = x$$

and let us write $\triangleright$ for the corresponding reduction relation.

**Fact.** $([x]M\ N) \triangleright^* M[x \leftarrow N]$.

We leave the proof of this very important property to the reader. The important point is that the abstraction operation, together with the application operator and the reduction $\triangleright$, define a *substitution* machinery. We shall now use this idea more generally, in order to internalize the deduction theorem in a basic calculus of functionality. That is, we forget the specific combinators $S$ and $K$, in favor of abstraction seen now as a new term constructor.

**Remark 1.** *Other abstraction operations may be defined. For instance, the* strong *abstraction algorithm is more economical:*

$$\begin{array}{ll} [x]x = I & \\ [x]M = K\ M & \text{if } x \text{ does not occur in } M \\ [x](M\ x) = M & \text{if } x \text{ does not occur in } M \\ [x](M\ N) = S\ [x]M\ [x]N & \text{otherwise.} \end{array}$$

**Remark 2.** The computation relation $\triangleright$ of combinatory algebra is confluent. Actually, it is defined by a particularly simple case of necessarily sequential rewrite rules. It is compatible with the term structure of combinatory algebra, and in particular with application. But it is not compatible with the derived operation of abstraction, and thus the $\xi$ rule of $\lambda$-conversion is not valid. That is, combinatory computation simulates only weak $\beta$-reduction.

Similarly to $\lambda$-calculus, there are typed and untyped versions of combinatory algebra.

Other combinators than $K$, $S$ and $I$ have been considered. A general combinator is defined by a rewrite rule:

$$C\ x_1\ x_2\ \ldots\ x_n\ :=\ M,$$

where the left-hand side stands for the pattern $App(\cdots App(C, x_1)\cdots, x_n)$ and the right-hand side is an arbitrary term constructed from the $x_i$'s, $App$, and previously defined combinators.

A set of combinators is said to form a *basis* if it is sufficient to derive an abstraction algorithm (equivalently, if $S$ and $K$ are definable from the set). The state of the art about combinatory completeness is described in Statman [174].

## 3.3 Typed Lambda-calculus

We now abandon the first-order term structures of combinatory algebra and turn to $\lambda$-calculi. We first consider *typed* $\lambda$-calculus, where the set of types $\mathcal{T}$ is defined as the set of terms constructed over some functor alphabet $\Phi$ containing the binary functor $\Rightarrow$. We write $\mathcal{T}^*$ for the set of finite sequences of types, with 1 the empty sequence and $\Gamma \times A$ the sequence obtained from sequence $\Gamma$ by adding one more type $A$.

We define recursively a relation $\Gamma \vdash M : A$, read "*$M$ is a term of type $A$ in context* $\Gamma$", where $A \in \mathcal{T}$ and $\Gamma \in \mathcal{T}^*$, as follows:

**Variable** : If $1 \leq n \leq |\Gamma|$ then $\Gamma \vdash n : \Gamma_n$
**Abstraction** : If $\Gamma \times A \vdash M : B$ then $\Gamma \vdash [A]M : A \Rightarrow B$
**Application** : If $\Gamma \vdash M : A \Rightarrow B$ and $\Gamma \vdash N : A$ then $\Gamma \vdash (M\ N) : B$

Thus a term may be a natural number, or may be of the form $[A]M$ with $A$ a type and $M$ a term, or may be of the form $(M\ N)$ with $M, N$ two terms.

We thus obtain typed $\lambda$-terms with variables coded as de Bruijn's indexes [16], i.e. as integers denoting their reference depth (distance in the tree to their binder). This representation avoids all the renaming problems associated with actual names ($\alpha$ conversion), but we shall use such names whenever we give examples of terms. For instance, the term $[A](1\ [B](1\ 2))$ shall be presented under a concrete representation such as $[x : A](x\ [y : B](y\ x))$. In Church's original notation, the left bracket was a $\lambda$ and the right bracket a dot, typing being indicated by superscripting, like: $\lambda x^A \cdot (x\ \lambda y^B \cdot (y\ x))$.

Note that the relation $\Gamma \vdash M : A$ is functional, in that $A$ is uniquely determined from $\Gamma$ and $M$. Thus the definition above may be interpreted as the recursive definition of a function $A\ =\ \tau_\Gamma(M)$.

The set $\mathcal{T}$ of types used in the $\lambda$-terms has been defined as all terms constructed from $\Phi$ containing $\Rightarrow$. The ordinary Curry-Church $\lambda$-calculus is obtained when $\Phi = \{\Rightarrow\} \cup \mathcal{T}_0$, where $\mathcal{T}_0$ is a finite set of atomic types, for instance $\{bool, int\}$. But we may include other functors in $\Phi$. The proofs of the intuitionistic version $NK$ of Gentzen's natural deduction system may be represented by typed $\lambda$-terms, over the alphabet of functors defined by the propositional connectives.

### 3.4 Computation

We are now ready to define the *computation* relation $\triangleright$ as follows:

$$([A]M\ N) \triangleright M\{N\} \qquad (\beta)$$

$$M \triangleright M' \Longrightarrow [A]M \triangleright [A]M' \qquad (\xi)$$

$$M \triangleright M' \Longrightarrow (M\ N) \triangleright (M'\ N)$$

$$M \triangleright M' \Longrightarrow (N\ M) \triangleright (N\ M').$$

It is clear that computation preserves the types of terms. The computation relation presented above is traditionally called (strong) $\beta$-reduction. It is confluent and nœtherian (because of the types!), and thus every term possesses a canonical form, obtainable by iterating computation non-deterministically. Another valid conversion rule is $\eta$-conversion:

$$[x:A](M\ x) = M \qquad (\eta)$$

whenever $x$ does not appear in $M$.

### 3.5 Weak reduction

There are many variations on $\lambda$-calculus. What we have just presented is typed $\lambda$-calculus, with Curry-Church types. The notion of computation $\triangleright$ is strong $\beta$ reduction. It is also interesting to consider a weak reduction, obtained by not allowing rule $\xi$ above. Thus, weak reduction is not compatible with the abstraction operator $\square$. As we have already seen, $\lambda$-calculus may be translated into combinatory algebra, but the natural computation rule associated with the set of combinator definitions seen as term rewriting system corresponds then to weak reduction, *not* strong reduction.

### 3.6 Pure $\lambda$-calculus

If we remove the types, we get the theory of pure $\lambda$-calculus. The set of pure lambda terms is defined as:

$$\lambda = \bigcup_{n\geq 0} \lambda_n$$

where the set $\lambda_n$ of $\lambda$-terms with $n$ potential free variables is defined inductively by:

- $i \in \lambda_n$ if $1 \leq i \leq n$
- $\square M \in \lambda_n$ if $M \in \lambda_{n+1}$
- $(M\ N) \in \lambda_n$ if $M, N \in \lambda_n$

As we did previously, we get readable concrete syntax by sticking variable names in the brackets, as in $[x]x$. The terms in $\lambda_0$ are the *closed* pure $\lambda$-terms. Analogous untyped versions of the rules above define analogous computation rules. Sometimes syntactic properties are easier to prove in pure $\lambda$-calculus. For instance, the confluence property in typed calculi is an easy consequence of the corresponding property in the pure calculus, if we remark that computation preserves typing. The classical method, due to Tait and Martin-Löf [4], consists in proving that the relation $\trianglerighteq$ is strongly confluent, with $\trianglerighteq$ defined as the reflexive and compatible closure of:

$$\frac{M \trianglerighteq M' \quad N \trianglerighteq N'}{(\square M\ N) \trianglerighteq M'\{N'\}}.$$

It is easy to check that indeed $\trianglerighteq$ and $\triangleright$ have the same reflexive-transitive closure, whence the result. As we saw for regular term rewriting system, such a "parallel moves" theorem is actually much stronger than strong confluence, since it corresponds to the existence of pushouts in an appropriate category of computations. The theory of $\lambda$-calculus derivations is worked out in detail in J.J. Lévy's thesis [108,109]. Note that contrarily to the theory of regular term rewriting systems, the parallel reduction $\trianglerighteq$ is not limited to parallel disjoint redexes, since in $\lambda$-calculus residuals of a redex may not be disjoint. For instance, consider $([u](u\ u)\ \ [v]([x]v\ y))$.

The theory of $\beta$-$\eta$-reduction is rather complicated. Actually, note that there is a critical pair between the two rules, since $([x](M\ x)\ N)$ contains conflicting redexes for the two rules. Fortunately, the two rules reduce to the same term $(M\ N)$. However, the two rules are usually dealt with separately, since it can be showed that $\eta$ conversions can be postponed after $\beta$ reductions. In the following, we write $\triangleright$ for the $\beta$-reduction rule, and $\equiv$ for its associated congruence. The theory of $\beta$-reduction is similar to the theory of regular term rewriting systems. Certain results are simpler. For instance, the standardization theorem has a simpler form, since the standard derivation always reduces the leftmost needed redex. Others are more complicated, due to the residual embedding noted earlier.

Certain theorems are identical for the pure calculus as for the typed case. Other aspects of pure $\lambda$-calculus differ from the typed version. In the pure calculus, some terms do not always admit normal forms. For instance, with $\Delta\ =\ [u](u\ u)$ and $\perp\ =\ (\Delta\ \Delta)$, we get $\perp\ \triangleright \perp\ \triangleright \ldots$ A more interesting example is given by

$$Y\ =\ [f]([u](f\ (u\ u))\ \ [u](f\ (u\ u)))$$

since $(Y\ M)\ \equiv\ (M\ (Y\ M))$ shows that $Y$ defines a general fixpoint operator. $Y$ is called the *Curry* fixpoint operator. Other fixpoint operators are known. For instance, the *Turing* fixpoint operator is defined as:

$$\Theta\ =\ ([x][y](y\ (x\ x\ y))\ \ [x][y](y\ (x\ x\ y))),$$

and it verifies the stronger property that for every $M$ we have $(\Theta\ M)\ \trianglerighteq^*\ (M\ (\Theta\ M))$.

**Exercise.** Show that $\Phi\ =\ [\varphi][f](f\ (\varphi\ f))$ is a generator of fixpoints, in that $M$ is a fixpoint combinator iff $\Phi(M) \equiv M$.

The existence of fixpoint operators, and the easy encoding of arithmetic notions in pure $\lambda$-calculus, make it a computationally complete formalism: all partial recursive functions are definable. We shall not develop further this aspect of $\lambda$-calculus, but we just remark that it entails the undecidability of most syntactic properties. Thus $\equiv$ is an undecidable relation, and it is generally undecidable whether a given term is normalisable or not.

What we are mostly concerned here is the application of $\lambda$-calculus to logic. And one may worry about the interpretation of fixpoints of propositional connectives such as negation. The next section shows that indeed pure $\lambda$-calculus is logically problematic.

### 3.7 Curry's version of Russell's paradox

Our framework is minimal logic, with propositions represented as pure $\lambda$-expressions. That is, we assume that $\Rightarrow$ is a constant of the calculus. We assume that we have as rules of inference:

$$A \Rightarrow B\ ,\ A\ \vdash\ B \qquad (App)$$

$$\vdash A \Rightarrow A \qquad (I)$$

$$\vdash (A \Rightarrow (A \Rightarrow B)) \Rightarrow (A \Rightarrow B) \qquad (W)$$

It is easy to see that $(W)$ is valid in minimal logic (consider $[u : A \Rightarrow (A \Rightarrow B)]\,[v : A](u\ v\ v)$). Now consider an arbitrary proposition $X$. Let us define $N = [A]A \Rightarrow X$, and let $M = (Y\ N)$. $N$ is in a way the minimal meaning for negation, and $M$ is a fixpoint of it. That is:

$$M \equiv (M \Rightarrow X). \qquad (*)$$

Now we get $M \Rightarrow M$ from $I_M$, and thus $M \Rightarrow (M \Rightarrow X)$ by $(*)$ used as an equality. Using *App* and $W$ we infer $M \Rightarrow X$, and thus $M$ using $(*)$ in the reverse direction. A final use of *App* yields $X$, which is an arbitrary proposition, and thus the logic is inconsistent [48].

Thus combinatory completeness of the pure $\lambda$-calculus at the level of propositions is not compatible with the logical completeness issued from the typed $\lambda$-calculus at the level of proofs.

Half way between the typed and the pure calculus we find typed calculi where additional constants and reduction rules have been added. For instance, it is possible to add typed recursion operators in order to develop recursive arithmetic in a sound way [175].

### 3.8 ML's polymorphism

We saw that formal systems could be pleasantly presented using polymorphic operators (inference rules) at the meta level. This possibility could be pushed at the user level, by allowing him to extend the system with derived polymorphic constants. We also saw that $\lambda$-calculus allowed the user to do proofs modulo a set of hypotheses $\Gamma$. However there is a fundamental difference between the apparent similarity between the notations $\Sigma \vdash ...$ and $\Gamma \vdash ....$ That is, when a constant declaration $C : \tau$ is in $\Sigma$ we allow it to be polymorphic, whereas when a variable declaration $x : \tau$ is in $\Gamma$ we request its type $\tau$ to be a constant term.

There is no immediate possible extension of polymorphism to variables, because the implicit universal quantification of type variables does not commute well with abstraction, because $\Rightarrow$ is contravariant on the left. We need to face up this problem by introducing some explicit quantification for type variables. A weak form of such polymorphism is implemented in ML, and explained below. A more general form will be explained in the section on polymorphic $\lambda$-calculus below.

This idea of type quantification corresponds to allowing proposition quantifiers in our propositional logic. First we allow a universal quantifier in prenex position. That is, with $T_0 = T(\Phi, V)$, we now introduce *type schemas* in $T_1 = T_0 \cup \forall\alpha \cdot T_1$, $\alpha \in V$. A (type) term in $T_1$ has thus both free and bound variables, and we write $FV(M)$ and $BV(M)$ for the sets of free (respectively bound) variables. We shall use systematically in the following the meta variables $\tau$, $\tau'$, etc... for type schemes in $T_1$, whereas un-quantified types from $T_0$ are denoted $\tau_0$, $\tau_0'$, etc...

We now define *generic instantiation*.
Let $\tau = \forall\alpha_1...\alpha_m \cdot \tau_0 \in T_1$ and $\tau' = \forall\beta_1...\beta_n \cdot \tau_0' \in T_1$. We define $\tau' \geq_G \tau$ iff $\tau_0' = \sigma(\tau_0)$ with $D(\sigma) \subseteq \{\alpha_1, ..., \alpha_m\}$ and $\beta_i \notin FV(\tau)$ $(1 \leq i \leq n)$. Note that $\geq$ acts on $FV$ whereas $\geq_G$ acts on $BV$. Also note

$$\tau' \geq_G \tau \quad \Rightarrow \quad \sigma(\tau') \geq_G \sigma(\tau).$$

We now present the Damas-Milner inference system for polymorphic $\lambda$-calculus [50]. In what follows, a sequent hypothesis $\Gamma$ is assumed to be a list of specifications $x_i : \tau_i$, with $\tau_i \in T_1$, and we write $FV(\Gamma) = \bigcup_i FV(\tau_i)$.

$$TAUT : \Gamma \vdash x : \tau \quad (x : \tau \in \Gamma)$$

$$INST : \frac{\Gamma \vdash M : \tau}{\Gamma \vdash M : \tau'} \quad (\tau \leq_G \tau')$$

$$GEN : \frac{\Gamma \vdash M : \tau}{\Gamma \vdash M : \forall\alpha\cdot\tau} \quad (\alpha \notin FV(\Gamma))$$

$$APP : \frac{\Gamma \vdash M : \tau'_0 \rightarrow \tau_0 \quad \Gamma \vdash N : \tau'_0}{\Gamma \vdash (M\ N) : \tau_0}$$

$$ABS : \frac{\Gamma \cup \{x : \tau'_0\} \vdash M : \tau_0}{\Gamma \vdash [x]M : \tau'_0 \rightarrow \tau_0}$$

$$LET : \frac{\Gamma \vdash M : \tau' \quad \Gamma \cup \{x : \tau'\} \vdash N : \tau}{\Gamma \vdash \mathit{let}\ x = M\ \mathit{in}\ N : \tau}.$$

Note that here the context $\Gamma$ stores both the variables (introduced with $ABS$) and the constants (introduced with $LET$). However constants are allowed to be polymorphic, whereas variables are limited to ordinary types from $T_0$.

**Example.** We get for instance:

$$\vdash \mathit{let}\ i\ =\ [x]x\ \mathit{in}\ (i\ i)\ :\ \alpha \rightarrow \alpha$$

whereas the term $[x](x\ x)$ cannot be typed in the system.

The above system may be extended without difficulty by other functors such as product, and by other ML constructions such as conditional, equality and recursion:

$$PROD : \frac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : \tau'}{\Gamma \vdash (M,N) : \tau \times \tau'}$$

$$FST : \Gamma \vdash \mathit{fst} : \forall\alpha\beta\cdot(\alpha \times \beta) \rightarrow \alpha$$

$$SND : \Gamma \vdash \mathit{snd} : \forall\alpha\beta\cdot(\alpha \times \beta) \rightarrow \beta$$

$$IF : \frac{\Gamma \vdash P : \mathit{bool} \quad \Gamma \vdash M : \alpha \quad \Gamma \vdash N : \alpha}{\Gamma \vdash \mathit{if}\ P\ \mathit{then}\ M\ \mathit{else}\ N : \alpha}$$

$$EQ : \Gamma \vdash = : \forall\alpha\cdot(\alpha \times \alpha) \rightarrow \mathit{bool}$$

$$REC : \Gamma \vdash Y : \forall\alpha\cdot(\alpha \rightarrow \alpha) \rightarrow \alpha$$

and we define $\mathit{let\ rec}\ x\ =\ M\ \mathit{in}\ N$ as an abbreviation for $\mathit{let}\ x\ =\ Y([x]M)\ \mathit{in}\ N$.

Every ML compiler contains a type-checker implementing implicitly the above inference system. For instance, with the unary functor *list* and the following ML primitives: $\Box$ : $(\mathit{list}\ \alpha)$, $\mathit{cons}$ : $\alpha \times (\mathit{list}\ \alpha)$ (written infix as a dot), $\mathit{hd} : (\mathit{list}\ \alpha) \rightarrow \alpha$ and $\mathit{tl} : (\mathit{list}\ \alpha) \rightarrow (\mathit{list}\ \alpha)$, we may define recursively the map functional as:

$$\mathit{let\ rec\ map}\ f\ l\ =\ \mathit{if}\ l = \Box\ \mathit{then}\ \Box\ \mathit{else}\ (f\ (\mathit{hd}\ l))\ \cdot\ \mathit{map}\ f\ (\mathit{tl}\ l)$$

and we get as its type:

$$\vdash\ \mathit{map} : (\alpha \rightarrow \beta) \rightarrow (\mathit{list}\ \alpha) \rightarrow (\mathit{list}\ \beta).$$

Of course the ML compiler is not implemented directly from the inference system above, which is non-deterministic because of rules $INST$ and $GEN$. It uses unification instead, and thus computes deterministically a principal type, which is minimum with respect to $\leq_G$:

**Milner's Theorem.** Every typable expression of the polymorphic $\lambda$-calculus possesses a principal type, minimum with respect to generic instantiation [129].

ML is a strongly typed programming language, where type inference is possible because of the above theorem: the user need not write type specifications. The compiler of the language does more than type-checking, since it actually performs a proof synthesis. Types disappear at run time, but because of the type analysis no dynamic checks are needed to enforce the consistency of data operations, and this allows fast execution of ML programs. ML is actually a generic name for languages of the ML family. For instance, by adding exceptions, abstract data types (permitting in particular user-defined functors) and references, one gets approximately the meta-language of the LCF proof assistant [66]. By adding record type declarations (i.e. labeled sums and products) one gets L. Cardelli's ML [22]. By adding constructor types, pattern-matching and concrete syntax, we get the ML presented in Chapter 1. A more complete language, including modules, is under design as Standard ML [130]. Current research topics on the design of ML -like languages are the incorporation of object-oriented features allowing subtypes, remanent data structures and bitmap operations [23], and "lazy evaluation" permitting streams and ZF expressions [187,138].

Note on the relationship between ML and $\lambda$-calculus. First, ML uses so-called call by value implementation of procedure call, corresponding to innermost reduction, as opposed to the outermost regime of the standard reduction. Lazy evaluation permits standard reductions, but closures (i.e. objects of a functional type $\alpha \rightarrow \beta$) are *not* evaluated. Finally, types in ML serve for ensuring the integrity of data operations, but still allow infinite computations by *non-terminating* recursions.

**Remark1.** The complexity of ML's type computation algorithm has been recently analysed by Kanellakis and Mitchell[92]. Rather surprisingly, the problem was shown to be PSPACE hard. This stands in contrast to the linear time algorithm which may be used to compute principal types in the Principal Type Theorem above. This may be explained intuitively as follows. Lambda expressions typable in the simple type discipline possess a principal type which is computable in linear time, like for combinatory logic above. ML's polymorphism, and typing complexity, arises from the *let* construct. Intuitively, *let* expressions are marked redexes, whose parallel reduction is simulated by ML's typing algorithm. The exponential factor comes from a possible blow-up in the size of the corresponding reduced term, due to embeddings of *let*'s. This potential exponential blow-up does not seem to be practically problematical, since programmers do not usually write expressions with a high level of *let* nesting.

**Remark2.** The typing rule for recursion is not as general as one might wish, since the bound recursive variable may not be used polymorphically inside the body. We may rather define *let rec* $x = M$ *in* $N$ as an abbreviation for *let* $x = \mu x \cdot M$ *in* $N$, where the $\mu$ binding operator obeys the typing rule:

$$MU \; : \; \frac{\Gamma \cup \{x : \tau\} \vdash M \; : \; \tau}{\Gamma \vdash \mu x \cdot M \; : \; \tau}.$$

With this new convention, we may now typecheck terms such as:

$$let\ K = [x]\,[y]x \quad in \quad let\ rec\ F = [x](F\ (K\ x)).$$

However, it is not known whether such an extended system admits a principal typing algorithm[132] (and even whether type-checking stays indeed decidable.)

### 3.9 The limits of ML 's polymorphism

Consider the following ML definition:

$$let\ rec\ power\ n\ f\ u\ =\ if\ n = 0\ then\ u\ else\ f\ (power\ (n-1)\ f\ u)$$

of type $nat \rightarrow (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$. This function, which associates to natural $n$ the polymorphic iterator mapping function $f$ to the $n$-th power of $f$, may be considered a coercion operator between ML 's internal naturals and Church's representation of naturals in pure $\lambda$-calculus [30]. Let us recall briefly this representation. Integer 0 is represented as the projection term $[f]\,[u]u$. Integer 1 is $[f]\,[u](f\ u)$. More generally, $n$ is represented as the functional $\bar{n}$ iterating a function $f$ to its $n$-th power:

$$\bar{n}\ =\ [f]\,[u](f\ (f\ ...(f\ u)...))$$

and the arithmetic operators may be coded respectively as:

$$n + m\ =\ [f]\,[u](n\ f\ (m\ f\ u))$$

$$n \times m\ =\ [f](n\ (m\ f))$$

$$n^m\ =\ (m\ n).$$

For instance, with $\bar{2}\ =\ [f]\,[u](f\ (f\ u))$, we check that $\bar{2} \times \bar{2}$ converts to its normal form $\bar{4}$.

We would like to consider a type

$$NAT\ =\ \forall\alpha \cdot (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$$

and be able to type the operations above as functions of type $NAT \rightarrow NAT \rightarrow NAT$. However the notion of polymorphism found in ML does not support such a type, it allows only the weaker

$$\forall\alpha \cdot ((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)) \rightarrow ((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)) \rightarrow ((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))$$

which is inadequate, since it forces the *same* generic instantiation of $NAT$ in the two arguments.

## 4 Polymorphic λ-calculus

The example above suggests using the universal type quantifier *inside* type formulas. We thus consider a functor alphabet based on one binary → constructor and one quantifier ∀. We shall now consider a λ-calculus with such types, which we shall call *second-order* or *polymorphic λ-calculus*, owing to the fact that the type language is now a second-order propositional logic, with propositional variables explicitly quantified. In order to emphasize this connection, we actually write ⇒ instead of →. In this calculus, we shall be able to form types (propositions) such as:

$$(\forall A \cdot A \Rightarrow A) \Rightarrow (\forall A \cdot A \Rightarrow A).$$

Such a calculus was proposed by J.Y. Girard [61,62], and independently discovered by J. Reynolds [155].

### 4.1 The inference system

We now have two kinds of variables, the variables bound by $\lambda$-abstraction, and the propositional variables. Each kind will have its own de Bruijn indexing scheme, but we put both kinds of bindings in one context sequence, in order to ensure that in a $\lambda$-binding $[x : P]$ the free propositional variables of $P$ are correctly scoped. A *context* $\Gamma$ is thus a sequence of bindings $[x : P]$ and of bindings $[A : Prop]$. We use de Bruijn indexes $V(n)$ and $P(n)$ to reference respectively the two kinds of variables. However, there is a slight difficulty if one tries to adhere too strictly to de Bruijn's notation. Consider the context $\Gamma = [A : Prop]\,[x : A]\,[B : Prop]$. In concrete syntax, we write $\Gamma \vdash x : A$. But if we use de Bruijn's indexes for propositional names, we get in the abstract syntax $\Gamma \vdash V(1) : P(2)$, i.e. the propositions have to be relocated.

In order to remedy this notational difficulty, we shall assume a mixed naming scheme, allowing concrete names for free variables of expressions as well as integers for bound variables. The binding operation $[x : P]M$ denotes now the abstract $[P]M'$, where $M'$ is $M$ where every occurrence of $x$ is replaced by the correct de Bruijn's index. Similarly we provide a binding operation $\forall A \cdot P$ for propositional variables. Finally an operation $\Lambda A \cdot M$ binds a propositional variable in a term.

A context $\Gamma$ is said to be *valid* if it binds variables with well-formed propositions. Thus the empty context is valid, if $\Gamma$ is valid and does not bind $A$ then $\Gamma[A : Prop]$ is valid, and finally if $\Gamma$ is valid and does not bind $x$ then $\Gamma[x : P]$ is valid provided $\Gamma \vdash P : Prop$. This last judgement (propositional formation) is defined recursively as follows:

$$\frac{[A : Prop] \in \Gamma}{\Gamma \vdash A : Prop}$$

$$\frac{\Gamma \vdash P : Prop \qquad \Gamma \vdash Q : Prop}{\Gamma \vdash P \Rightarrow Q : Prop}$$

$$\frac{\Gamma[A : Prop] \vdash P : Prop}{\Gamma \vdash \forall A \cdot P : Prop}.$$

Let us now give the term-formation rules. We have two more constructors: $\Lambda A \cdot M$ which makes a term polymorphic, by $\forall$-introduction, and $< M\ P >$, which instantiates the polymorphic term $M$ over the type corresponding to proposition $P$, by $\forall$-elimination.

$$\begin{aligned}
Var &: \frac{[x\,.\,P] \in \Gamma}{\Gamma \vdash x : P}\\
Abstr &: \frac{\Gamma \vdash P : Prop \qquad \Gamma[x : P] \vdash M : Q}{\Gamma \vdash [x : P]M : P \Rightarrow Q}\\
Appl &: \frac{\Gamma \vdash M : P \Rightarrow Q \qquad \Gamma \vdash N : P}{\Gamma \vdash (M\ N) : Q}\\
Gen &: \frac{\Gamma[A : Prop] \vdash M : P}{\Gamma \vdash \Lambda A \cdot M : \forall A \cdot P}\\
Inst &: \frac{\Gamma \vdash M : \forall A \cdot P \qquad \Gamma \vdash Q : Prop}{\Gamma \vdash < M\ Q >: P\{Q\}_P}.
\end{aligned}$$

We do not make explicit the propositional substitution operation $P\{Q\}_P$, which is defined similarly to the $\lambda$-calculus substitution $M\{N\}$ seen previously. The latter will be denoted here by $M\{N\}_V$.

**Proposition 1.** If $\Gamma$ is valid, then $\Gamma \vdash M : P$ implies $\Gamma \vdash P : Prop$.

We leave the proof of such easy (but tedious) lemmas to the patient reader.

Let us now give an example of a derivation. Let $Id := \Lambda A \cdot [x : A]x$. $Id$ is the polymorphic identity algorithm, and we check easily that $\vdash Id : One$, where $One := \forall A \cdot A \Rightarrow A$. Note that indeed $One$ is well-formed in the empty context. Now we may instantiate $Id$ over its own type $One$, yielding: $\vdash < Id\ One >: One \Rightarrow One$. The resulting term may thus be applied to $Id$, yielding: $\vdash (< Id\ One >\ Id) : One$.

Similarly, we can define a composition operator for proofs, whose type is the analogue of the cut, or detachment rule:
$[P : Prop][Q : Prop][R : Prop] \vdash [f : P \Rightarrow Q][g : Q \Rightarrow R][x : P](g\ (f\ x)) : ((P \Rightarrow Q) \Rightarrow (Q \Rightarrow R) \Rightarrow (P \Rightarrow R))$.
We shall use the notation $f;g$ as a shorthand for the too cumbersome $(< Compose\ P\ Q\ R >\ f\ g)$, since the type arguments $P$, $Q$ and $R$ can be retrieved as subparts of the types of $f$ and $g$.

## 4.2 The conversion rules

The calculus admits two conversion rules. The first one is just $\beta$:

$$\beta \ : \ \frac{}{\Gamma \vdash ([x : P]M\ N) \rhd M\{N\}_V}.$$

The second one eliminates the cut formed by introducing and eliminating a quantification:

$$\beta' \ : \ \frac{}{\Gamma \vdash < \Lambda A \cdot M\ P > \rhd M\{P\}_P}.$$

Of course, we assume all other rules extending $\rhd$ as a term congruence, as usual. We may also consider analogues of the $\eta$ rule.

**Proposition 2.** If $\Gamma$ is valid, $\Gamma \vdash M : P$ and $\Gamma \vdash M \rhd N$ then $\Gamma \vdash N : P$.

## 4.3 The syntactic interpretation

We proceed as in the last chapter. However, here there are no primitive types. In order to have a non-trivial interpretation, we introduce a supplementary constant $\Omega$ to our untyped $\lambda$-terms. Let $\lambda_\Omega$ be the set of such terms, and $SN$ be the set of strongly normalizable terms of $\lambda_\Omega$.

**Definition.** A subset $S$ of $SN$ is said to be *saturated* iff

- $\forall N \in SN \quad (M\{N\}\ M_1\ \ldots\ M_n) \in S \Rightarrow (\Box M\ N\ M_1\ \ldots\ M_n) \in S$
- $\Omega \in S$
- $N_1, \ldots, N_k \in SN \Rightarrow (\Omega\ N_1 \ldots N_k) \in S$.

Note that in the first clause, we may limit ourselves to considering $M$ and the $M_i$'s in $SN$. We write $Sat$ for the set of saturated subsets of $SN$.

We now define the interpretation $\mathcal{I}$ by defining for every term $M$ its corresponding pure term $\mathcal{I}(M) = \nu(M)$, where

- $\nu(V(n)) = n$
- $\nu([x : P]M) = \Box\nu(M)$

- $\nu((M\ N)) = (\nu(M)\ \nu(N))$
- $\nu(\Lambda A \cdot M) = \nu(M)$
- $\nu(<M\ P>) = \nu(M)$.

Note that $\nu(M)$ is a pure $\lambda$-term constructed over the list of free variables $\{x \mid [x : P] \in \Gamma\}$.

Finally, to every $A$ such that $[A : Prop] \in \Gamma$ we associate an arbitrary saturated set $\mathcal{I}(A)$. Let $I(\Gamma)$ be the product of all such $\mathcal{I}(A)$'s. We define recursively the interpretation $\mathcal{I}_{\mathcal{I}(\Gamma)}(P)$ of a proposition $P$, such that $\Gamma \vdash P : Prop$, as follows:

- $\mathcal{I}_G(P \Rightarrow Q) = \{M \mid \forall N \in \mathcal{I}_G(P) \quad (M\ N) \in \mathcal{I}_G(Q)\}$
- $\mathcal{I}_G(\forall A \cdot P) = \bigcap_{S \in Sat} \mathcal{I}_{G \times S}(P)$
- $\mathcal{I}_G(A) = G_A$.

**Example.** $\mathcal{I}(Id) = \square 1$. $\mathcal{I}(One)$ contains all strongly normalizable terms whose canonical form is $\square 1$, plus strongly normalizable terms whose canonical form has head variable $\Omega$.

### 4.4 Basic meta-mathematical properties

The main use of the interpretation above is to prove:

**Girard's theorem.** If $\Gamma$ is valid and $\Gamma \vdash M : P$, then $\mathcal{I}(M) \in \mathcal{I}(P) \in Sat$.
**Corollary 1.** $\nu(M) \in SN$.

**Corollary 2: Strong normalization.** The conversion $\triangleright$ on typed terms is Nœtherian.

(Note that $\beta'$ alone is Nœtherian).

**Definition.** Let $\Gamma$ be a valid context, with $\Gamma \vdash P : Prop$. We say that $P$ is *inhabited* in $\Gamma$ iff $\mathcal{I}_{\mathcal{I}(\Gamma)}(P)$ contains a term without $\Omega$'s.

Note that if $\Gamma \vdash M : P$, then $P$ is inhabited (by $\mathcal{I}(M)$). We know obtain the consistency of the logical system as:

**Soundness Theorem.** The type $\nabla = \forall A \cdot A$ is not inhabited.
**Corollary.** There is no term $M$ which proves $\nabla$.

**Undecidability Theorem (Löb).** The following problem is recursively unsolvable: Given a valid context $\Gamma$ and a proposition $P$, with $\Gamma \vdash P : Prop$, find whether or not there exists an $M$ such that $\Gamma \vdash M : P$.

The second-order $\lambda$-calculus does not admit principal types. For instance, we shall show below that the combinator $K$ may be typed in several incompatible manners. We may still wonder whether it is decidable whether an arbitrary pure $\lambda$-term is typable in the system or not. This is an important open problem:

**Problem.** Give a procedure which, given a pure $\lambda$-term $T$, decides whether or not there exist $M$ and $P$ such that $\vdash M : P$, with $T = \nu(M)$. Alternatively, show that the problem is undecidable.

## 4.5 Examples of polymorphic proofs

In this section, we demonstrate the power of expression of the second-order calculus by way of examples.

### 4.5.1 Intuitionistic connectives

We first show that the other propositional connectives are definable in the calculus. It is well known that the intuitionistic connectives are definable in the second-order propositional calculus. The encoding of conjunction was already proposed by Russell, as explained in Prawitz [150].

Let $P$ and $Q$ be two propositions. We define $P \wedge Q$ as the proposition:

$$P \wedge Q \;:=\; \forall A \cdot (P \Rightarrow Q \Rightarrow A) \Rightarrow A.$$

As usual, we associate implications to the right, and applications to the left. The definition above is a correct encoding of $\wedge$, as can be seen from the derivation of the standard rules of conjunction:

$$[P : Prop]\,[Q : Prop]\,[x : P]\,[y : Q] \vdash \Lambda A \cdot [h : P \Rightarrow Q \Rightarrow A](h\; x\; y) : P \wedge Q$$

$$[P : Prop]\,[Q : Prop]\,[x : P \wedge Q] \vdash (<x\; P> \;\; [u : P]\,[v : Q]\,u) : P$$

$$[P : Prop]\,[Q : Prop]\,[x : P \wedge Q] \vdash (<x\; Q> \;\; [u : P]\,[v : Q]\,v) : Q.$$

In order to understand this sort of definition, it is best to wonder what is the *operational* use of the concept one is trying to define. Once this is clear, the concept can be easily *programmed*. This *procedural interpretation* is faithful to the intuitionistic semantics. For instance, $P \wedge Q$ is a method for proving any proposition $A$, provided one has a proof that $A$ follows from $P$ and $Q$. Note that the proof of $\wedge$-introduction above is a pairing algorithm, the two projections being the proofs of $\wedge$-elimination on the left and on the right.

We may similarly "program" the (intuitionistic) sum $P + Q$ of two propositions $P$ and $Q$:

$$P + Q \;:=\; \forall A \cdot (P \Rightarrow A) \Rightarrow (Q \Rightarrow A) \Rightarrow A.$$

Sum elimination is proved by the conditional, or *case* expression:

$$[P : Prop]\,[Q : Prop] \vdash \Lambda A \cdot [u : P \Rightarrow A]\,[v : Q \Rightarrow A]\,[x : P + Q](<x\; A> \;\; u\; v)$$

$$: \forall A \cdot (P \Rightarrow A) \Rightarrow (Q \Rightarrow A) \Rightarrow (P + Q) \Rightarrow A.$$

The two sum introductions correspond to the two injections:

$$[P : Prop]\,[Q : Prop] \vdash [x : P]\,\Lambda A \cdot [u : P \Rightarrow A]\,[v : Q \Rightarrow A](u\; x) : P \Rightarrow (P + Q)$$

$$[P : Prop]\,[Q : Prop] \vdash [y : Q]\,\Lambda A \cdot [u : P \Rightarrow A]\,[v : Q \Rightarrow A](v\; y) : Q \Rightarrow (P + Q).$$

### 4.5.2 Classical logic

Classical reasoning is reasoning by contradiction. The contradiction, or absurd proposition, proves every proposition $A$ by mere application:

$\nabla \;:=\; \forall A \cdot A.$

$\nabla$ has no proof, and may thus play the rôle of the truth-value *False*. Negating a proposition amounts to asserting that it implies $\nabla$, whence the concept of negation:

$\neg\, [A : Prop] \;:=\; A \Rightarrow \nabla.$

The Sheffer's stroke $A \mid B$ (read "$A$ contradictory with $B$") may be defined as:
$[A : Prop] \mid [B : Prop] \;:=\; A \Rightarrow B \Rightarrow \nabla.$

It is easy to show $\forall A \cdot \forall B \cdot (A \mid B) \iff \neg(A \wedge B)$. The other classical connectives may be simply expressed in term of $\mid$ :
$[A : Prop] \supset [B : Prop] \;:=\; A \mid \neg B$
$[A : Prop] \vee [B : Prop] \;:=\; (\neg A) \mid (\neg B)$
$[A : Prop] \equiv [B : Prop] \;:=\; (A \supset B) \wedge (B \supset A).$

Let us call *classical closure* of proposition $A$ its double negation:
$C([A : Prop]) \;:=\; \neg(\neg A).$

Every proposition denies its negation:

$$[A : Prop] \vdash [p : A]\,[q : \neg A](q\ p) : A \Rightarrow C(A).$$

The reverse implication holds only of classical propositions:
$Classical([A : Prop]) \;:=\; C(A) \Rightarrow A.$

We can show that $\nabla, \neg\,, \mid$ construct only classical propositions, and thus so do $\vee$ and $\supset$. Finally, $\wedge$ preserves the property of being classical, and thus $\equiv$ constructs also classical propositions.

Actually, classical reasoning consists in general in showing that a set of propositions $\{A_1, ..., A_n\}$ is contradictory. The connectives $\nabla, \neg\,, \mid$ express this notion for $n = 0, 1, 2$ respectively.

Let us remark that it is easy to prove the principe of the excluded middle:
$[A : Prop] \vdash < Id\ C(A) >: \neg A \vee A.$

**Remark.** Many other encodings of the propositional connectives may be used. Let us give two alternate definitions for classical disjunction:
$[A : Prop] \vee' [B : Prop] \;:=\; C(A + B)$
$[A : Prop] \vee'' [B : Prop] \;:=\; \forall C \cdot Classical(C) \Rightarrow (A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C.$

We now turn to axiomatizing universal algebra and abstract data types.

### 4.5.3 Initial algebras

We first show how to formalize the elementary notions from Algebra, in particular the notion of free algebra over a given signature. We start with the homogeneous case, that is we assume in the following that contexts start with a proposition letter taken as unique sort: $[A : Prop]$.

For every $n \geq 0$, we define the *A-cardinal* $\overline{n}$ associated to $n$ by induction:

$\overline{0} = A$

$\overline{n+1} = A \Rightarrow \overline{n}.$

We define now the *functionality* $\varphi(\Sigma)$ associated to a signature $\Sigma$ represented as a list of operators given with their arity, by:
$\varphi(\emptyset) = A$
$\varphi([F : n]\ \Sigma) = \overline{n} \Rightarrow \varphi(\Sigma).$

Such definitions are easily programmable in the meta-language.

We now obtain the *weakly initial algebra* associated to signature $\Sigma$ by abstracting over the type given as carrier of the algebra:
$I(\Sigma) = \forall A \cdot \varphi(\Sigma)$.

Let us now consider an arbitrary $\Sigma$-algebra. That is, we assume we place ourselves in context $\Gamma$ :

$$\Gamma = [A : Prop]\,[F_1 : \overline{n_1}] \cdots [F_s : \overline{n_s}].$$

If $M : I(\Sigma)$ is an arbitrary construction of an element of the initial $\Sigma$-algebra, we call *image* of $M$ in the $\Sigma$-algebra $\Gamma$ the term $M^\Gamma = (<M\ A>\ \ F_1 \ \cdots\ F_s)$. We remark that this term is well-formed, with type $A$. This notion of image corresponds, classically, to taking the image of $M$ by the unique $\Sigma$-morphism from $I(\Sigma)$ to $\Gamma$. For instance, when $M_1 : I(\Sigma)$, $M_2 : I(\Sigma)$, ... , $M_{n_k} : I(\Sigma)$, we get $(F_k\ M_1^\Gamma \ \cdots\ M_{n_k}^\Gamma) : A$. We define thus a $F_k$ operator of arity $n_k$ over $I(\Sigma)$, that we call the $F_k$-constructor, obtained in discharging $\Gamma$, and a list of $n_k$ variables of type $I(\Sigma)$.

**Definition.** Let $\Sigma$ be an arbitrary signature of length $s$:

$$\Sigma = [F_1 : \overline{n_1}] \cdots [F_s : \overline{n_s}].$$

We define the set $Dat(\Sigma)$ of *data elements* of $\Sigma$ as the set:

$$\{\nu(M) \mid M = \Lambda A \cdot [F_1 : \overline{n_1}] \cdots [F_s : \overline{n_s}]\, N \ \text{ with } N \text{ canonical}\}.$$

**Remark.** The set of canonical elements in $\mathcal{I}(\Sigma)$ has too much redundancy if we do not assume the $\eta$ rule of conversion. The data elements restrict consideration to the $\lambda$-terms in $\eta$-expanded normal form:

**The Representation Theorem.** $Dat(\Sigma)$, structured with the constructors, is isomorphic to the initial algebra in the class of all $\Sigma$-algebras.

**Problem.** Prove the theorem above.

### 4.5.4 Examples of data types

Let us now give a few examples. When $\Sigma = \emptyset$, we get $I(\Sigma) = \nabla$, the empty algebra. When $\Sigma = [i : 0]$, we get $I(\Sigma) = One := \forall A \cdot A \Rightarrow A$, and the $i$-constructor is $Id = \Lambda A \cdot [i : A]i$.

With $\Sigma = [t : 0]\,[f : 0]$, we get: $I(\Sigma) = Bool := \forall A \cdot A \Rightarrow A \Rightarrow A$, and the two constructors are the Booleans of Church [30]:
$True := \Lambda A \cdot [t : A]\,[f : A]\ t$
$False := \Lambda A \cdot [t : A]\,[f : A]\ f$.

When $\Sigma = [s : 1]\,[z : 0]$, we get $I(\Sigma) = Nat$, Church's naturals :
$Nat := \forall A \cdot (A \Rightarrow A) \Rightarrow A \Rightarrow A$
$S := [n : Nat]\Lambda A \cdot [s : A \Rightarrow A]\,[z : A]\ (s\ (<n\ A>\ \ s\ z))$
$0 := \Lambda A \cdot [s : A \Rightarrow A]\,[z : A]\ z$.

When $\Sigma = [c : 2]\,[n : 0]$, we get $I(\Sigma) = Bin$, the binary trees:
$Bin := \forall A \cdot (A \Rightarrow A \Rightarrow A) \Rightarrow A \Rightarrow A$
$Cons := [a_1 : Bin]\,[a_2 : Bin]\Lambda A \cdot [c : A \Rightarrow A \Rightarrow A]\,[n : A]\ (c\ (<a_1\ A>\ \ c\ n)\ (<a_2\ A>\ \ c\ n))$
$Nil := [A : Prop]\,[c : A \Rightarrow A \Rightarrow A]\,[n : A]\ n$.

### 4.5.5 Generalization to non-homogeneous algebras

It is straightforward to generalize these notions to the non-homogeneous case, introducing as many sorts as necessary. For instance, the list structure is axiomatized on two sorts $A$ and $B$ as follows:
$\mathit{List} \ := \ \forall A, B \cdot (A \Rightarrow B \Rightarrow B) \Rightarrow B \Rightarrow B.$

The operation of adding an element to a list is polymorphic. Let us consider the list schema, over proposition $A$:
$\mathit{List}\ A \ := \ \forall B \cdot (A \Rightarrow B \Rightarrow B) \Rightarrow B \Rightarrow B.$
We now define, in context $\Gamma = [A : \mathit{Prop}]$:
$\mathit{Add} \ := \ [x : A]\,[L : (\mathit{List}\ A)]\Lambda B \cdot [c : A \Rightarrow B \Rightarrow B]\,[e : B]\ (c\ x\ (<L\ B>\ \ c\ e))$
$: \forall A \cdot A \Rightarrow (\mathit{List}\ A) \Rightarrow (\mathit{List}\ A).$

We remark the analogy with ML 's list constructor. Here the empty list is doubly polymorphic:
$\mathit{Empty} \ := \ \Lambda A \cdot \Lambda B \cdot [c : A \Rightarrow B \Rightarrow B]\,[e : B]e \ \ : \mathit{List}.$

More generally, we may define all the data structures corresponding to free algebras. We remark that the corresponding propositions are restricted to degree 2, with the degree $\delta$ defined as:

- $\delta(A) = 0 \qquad (A\ \mathit{variable})$
- $\delta(\forall A \cdot M) = \delta(M)$
- $\delta(P \Rightarrow Q) = \mathit{max}\{1 + \delta(P), \delta(Q)\}.$

**Problem.** Generalize the Representation theorem above to the non-homogeneous case.

### 4.5.6 Second-order arithmetic

Let us give a few examples of programs over naturals. Addition is obtained by iterating successor:
$\mathit{Plus} \ := \ [m : \mathit{Nat}]\,[n : \mathit{Nat}](<n\ \mathit{Nat}>\ \ S\ m).$
Other definitions are possible. Multiplication is similarly obtained by iterating addition:
$\mathit{Times} \ := \ [m : \mathit{Nat}]\,[n : \mathit{Nat}](<n\ \mathit{Nat}>\ \ (\mathit{Plus}\ m)\ 0).$

We may also "see" our naturals as polymorphic iterators. Another possible definition of multiplication of $m$ and $n$ would thus be the composition $m;n$.

Exponentiation is obtained by iterating multiplication:
$\mathit{Exp} \ := \ [m : \mathit{Nat}]\,[n : \mathit{Nat}](<n\ \mathit{Nat}>\ \ (\mathit{Times}\ m)\ (S\ 0)).$

Iterating a natural on a functional type may produce non-primitive recursive functions; for instance we get Ackermann's function by diagonalization:
$\mathit{Ack} \ := \ [n : \mathit{Nat}](<n\ (\mathit{Nat} \Rightarrow \mathit{Nat})>\ \ ([f : \mathit{Nat} \Rightarrow \mathit{Nat}]\,[m : \mathit{Nat}]\ (<m\ \mathit{Nat}>\ \ f\ m))\ S).$

Indeed, most (total) recursive functions are definable as proofs in this formal system:

**Theorem.** (Girard [62]. See also [173]). Every recursive function provably total in second-order arithmetic is definable as a proof of type $\mathit{Nat} \Rightarrow \mathit{Nat}$ in the polymorphic $\lambda$-calculus.

### 4.5.7 Algebraic Programming

We may consider the polymorphic $\lambda$-calculus a powerful applicative programming language. It is both poorer than ML , in that no universal recursion operator is available, and richer, in that it provides a more complicated type structure. The price to pay is that there is no algorithm for synthesizing a principal type.

This language is revolutionary, in that it confuses *data structures* and *control structures*. Here, a data structure is but an unfulfilled control structure, waiting for more arguments to be able to "compute itself out". Thus to each of the data types seen above corresponds naturally a control structure. For *One* it is just the identity algorithm. For *Bool* it is the notion of conditional; that is, if $b : \mathit{Bool}$ and $M : A$, $N : A$ are the two branches of the conditional, the expression *If b Then M Else N* may be implemented as $(< b\ A >\ M\ N) : A$. For *Nat*, the polymorphic natural $n : \mathit{Nat}$ may be thought of as the construction `for i:= 1 to n do`. Compare this with `iterate n`, as defined in 1.1.1. Note that equality to zero is easily defined as:
$\mathit{EqZero} := [n : \mathit{Nat}](<n\ \mathit{Bool}>\ [b : \mathit{Bool}]\mathit{False}\ \mathit{True})$.

As remarked above, the conjunction connective builds in product. Writing alternatively $A \times B$ for $A \wedge B$ as defined above, we get the pairing and projection algorithms as proofs of *respectively* $\wedge$-intro and $\wedge$-elim:
$\mathit{Pair} := \Lambda A, B \cdot [x : A]\,[y : B]\Lambda C \cdot [h : A \Rightarrow B \Rightarrow C](h\ x\ y)$
$\mathit{Fst} := \Lambda A, B \cdot [x : A \times B](<x\ A>\ [u : A]\,[v : B]u)$
$\mathit{Snd} := \Lambda A, B \cdot [x : A \times B](<x\ B>\ [u : A]\,[v : B]v)$
Thus, for instance, for any types $A$ and $B$, $<\mathit{Fst}\ A\ B>: A \times B \Rightarrow A$, just as in ML .

However, the sum constructor is different: there is no analogue of the operators `outl` and `outr` here, since all the functions we may define are total:
$\mathit{Case} := \Lambda A, B \cdot [x : A + B]\Lambda C \cdot [u : A \Rightarrow C]\,[v : B \Rightarrow C](<x\ C>\ u\ v)$
$\mathit{Inl} := \Lambda A, B \cdot [x : A]\Lambda C \cdot [u : A \Rightarrow C]\,[v : B \Rightarrow C](u\ x)$
$\mathit{Inr} := \Lambda A, B \cdot [x : B]\Lambda C \cdot [u : A \Rightarrow C]\,[v : B \Rightarrow C](v\ x)$

### 4.5.8 Primitive recursion

It is possible to represent standard program schemas by combinators. For instance, it is shown in [39] how to define simple primitive recursive schemes.

### 4.5.9 Ordinals

All the propositions (types) considered above are very simple, since they are restricted to degree 2.

With more complex types, we may define richer data structures. For instance, Th. Coquand [36] has shown how to define ordinal notations, as an extension of the naturals above. We just enrich *Nat* with a limit operation, which associates an ordinal to a sequence of ordinals, represented as a function of domain *Nat*. We define thus:
$\mathit{Ord} := \forall A \cdot ((\mathit{Nat} \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow A) \Rightarrow A \Rightarrow A$
$\mathit{Olim} := [\sigma : \mathit{Nat} \Rightarrow \mathit{Ord}]\Lambda A \cdot [li : (\mathit{Nat} \Rightarrow A) \Rightarrow A]\,[s : A \Rightarrow A]\,[z : A](li\ [n : \mathit{Nat}](<(\sigma\ n)\ A>\ li\ s\ z))$
$\mathit{Osucc} := [\alpha : \mathit{Ord}]\Lambda A \cdot [li : (\mathit{Nat} \Rightarrow A) \Rightarrow A]\,[s : A \Rightarrow A]\,[z : A]\ (s\ (<\alpha\ A>\ s\ z))$
$\mathit{Ozero} := \Lambda A \cdot [li : (\mathit{Nat} \Rightarrow A) \Rightarrow A]\,[s : A \Rightarrow A]\,[z : A]\ z.$

It is straightforward to coerce a natural into the corresponding ordinal, which defines the sequence of finite ordinals:
$\mathit{Finite} := [n : \mathit{Nat}]\ (<n\ \mathit{Ord}>\ \mathit{Osucc}\ \mathit{Ozero})$.

Note that we instantiate the polymorphic natural $n$ over type *Ord*. Thus the meaning of type quantification is to quantify over an arbitrary proposition definable in the calculus, and not simply over some totality circumscribed to the construction at hand. In other words, the calculus is inherently non *predicative*, and we are using this feature in an essential way.

The first transfinite ordinal, $\omega$, may be simply obtained as limit of finite ordinals:
$\omega := (\mathit{Olim}\ \mathit{Finite})$.

We may program over ordinals the same way we do with naturals:
$\mathit{Oplus} := [\alpha : \mathit{Ord}][\beta : \mathit{Ord}](<\beta\ \mathit{Ord}>\ \mathit{Olim}\ \mathit{Osucc}\ \alpha)$
$\mathit{Otimes} := [\alpha : \mathit{Ord}][\beta : \mathit{Ord}](<\beta\ \mathit{Ord}>\ \mathit{Olim}\ (\mathit{Oplus}\ \alpha)\ \mathit{Ozero})$
$\mathit{Oexp} := [\alpha : \mathit{Ord}][\beta : \mathit{Ord}](<\beta\ \mathit{Ord}>\ \mathit{Olim}\ (\mathit{Otimes}\ \alpha)\ (\mathit{Osucc}\ \mathit{Ozero}))$.

Our ordinals are in fact *ordinal notations*, i.e. ordinals presented by fundamental sequences. In particular, $(\mathit{Oplus}\ (\mathit{Osucc}\ \mathit{Ozero})\ \omega)$ and $\omega$ are two distinct constructions.

We may get the ordinal $\epsilon_0$ as the iteration $(\mathit{Oexp}\ \omega\ (\mathit{Oexp}\ \omega \cdots))$:
$\epsilon_0 := (<\omega\ \mathit{Ord}>\ \mathit{Olim}\ (\mathit{Oexp}\ \omega)\ \mathit{Ozero})$.

We may now use ordinals to define functional hierarchies. First, we give preliminary definitions concerning integer functions:
$\mathit{Incr} := [f : \mathit{Nat} \Rightarrow \mathit{Nat}][n : \mathit{Nat}]\ (S\ (f\ n))$
$\mathit{Iter} := [f : \mathit{Nat} \Rightarrow \mathit{Nat}][n : \mathit{Nat}]\ (<n\ \mathit{Nat}>\ f\ n)$
$\mathit{Diag} := [\sigma : \mathit{Nat} \Rightarrow \mathit{Nat} \Rightarrow \mathit{Nat}][n : \mathit{Nat}]\ (\sigma\ n\ n)$.

Schwichtenberg's fast hierarchy may be defined as:
$\mathit{Fast} := [\alpha : \mathit{Ord}](<\alpha\ (\mathit{Nat} \Rightarrow \mathit{Nat})>\ \mathit{Diag}\ \mathit{Iter}\ \mathit{Osucc})$
and the slow hierarchy is defined similarly (note that we just change the successor argument):
$\mathit{Slow} := [\alpha : \mathit{Ord}](<\alpha\ (\mathit{Nat} \Rightarrow \mathit{Nat})>\ \mathit{Diag}\ \mathit{Incr}\ \mathit{Osucc})$.

It is to be noted that $(\mathit{Fast}\ \epsilon_0)$ is a total recursive function, but this fact is independent (i.e. undecidable) from Peano's arithmetic [58,100].

## 5 The Calculus of Constructions

### 5.1 Designing a higher-order system

The first step consists in extending the polymorphic $\lambda$-calculus in order to allow the binding of proposition *schemas*. This permits the definition of propositional connectives inside the formalism. For instance, in polymorphic $\lambda$-calculus, we defined $\wedge$ at the level of the meta-notation: $\wedge$ was just a macro of the meta language expanding into a proposition of the formal system. Now we want to be able to write $\wedge$ as a combinator internally.

Next we abstract on such propositional connectives, leading to a higher-order propositional calculus. The first problem we encounter is a notational one. We shall have to distinguish between the proposition schemas, where some variable is functionally abstracted, and the propositions where the same variable is universally quantified.

**Convention.** We shall keep the square brackets for functional abstraction, and use parentheses for universal quantification, using the traditional notation $(x : A)M$.

The second extension consists in adding a first-order part, allowing quantification and abstraction on "elements". The natural question to investigate is: what are we going to choose as the types of the elements? The simplest decision is to follow once more the Curry-Howard paradigm: we already have the proofs, as elements of the types the propositions. This gives us not only 1st-order logic, but higher-order logic as well, since an implication will play the role of a functional type, and thus we encompass Church's theory of types just because we shall have intuitionistic propositional calculus as a sub-system of the propositions. We may wonder why it is legitimate to use the proofs as elements: aren't we pre-supposing some *structure of our domains*? Actually not, since the proofs are the bare bones of a functional type system: they are nothing more and nothing less than the $\lambda$-expressions of the right type.

Let us thus assume that we have propositions closed under quantification $(x : P)Q$ and abstraction $[x : P]Q$. The first remark is that implication becomes a derived notion: $P \Rightarrow Q$ is just a notational variant for $(x : P)Q$ in the special case when $x$ does not occur in $Q$. What we shall now get is an intuitionistic version of Church's theory of types with dependent products.

## 5.2 The Calculus of Constructions, first version

### 5.2.1 The inference system

Let us now introduce explicitly a constant *Prop* for the type of propositions. At the level of proofs $[P : Prop]\,M$ gives us what we wrote previously $\Lambda P \cdot M$. Similarly, quantifying a proposition over *Prop*, as in $(P : Prop)Q$, gives us what we wrote previously $\forall P \cdot Q$. This suggests unifying also the notation $<M\ P>$ with $(M\ N)$. We thus arrive at a very simple calculus.

The types of proposition schemas are formed by quantification over the constant *Prop*. Let us use the constant *Type* for denoting all such types. We thus have two "kinds" of types: the types in the sense of Church's type theory, which here are all the terms of type *Type*, and the types in the sense of the propositions as types principle, which are here all the terms of type *Prop*. In the following, we use the meta-variable $K$ (for an arbitrary *kind*) to stand for either of the constants *Type* and *Prop*.

In all the following rules, $\Gamma$ is assumed to be a valid context, where the rules for valid contexts are:

- The *empty context* {} is valid.
- If $\Gamma$ is a valid context which does not bind variable $x$ and $\Gamma \vdash T : K$ then $\Gamma[x : T]$ is a valid context.
- If $\Gamma$ is a valid context which does not bind variable $t$ then $\Gamma[t : Type]$ is a valid context.

The first rule concerns accessing variables in a context:

$$Var : \frac{[x : T] \in \Gamma}{\Gamma \vdash x : T}.$$

The above rule is shorthand for $\Gamma \vdash Var(k) : T^{+(k-1)}$ when $\Gamma_k = [x : T]$.

We state that *Prop* is the only pre-defined atomic type:

$$Prop : \Gamma \vdash Prop : Type$$

More types are obtained by quantification, seen as generalized product:

$$Product : \frac{\Gamma \vdash P : K \quad \Gamma[A : P] \vdash M : Type}{\Gamma \vdash (A : P)M : Type}$$

Similarly, quantification on propositions gives more propositions:

$$Quant : \frac{\Gamma \vdash P : K \quad \Gamma[A : P] \vdash M : Prop}{\Gamma \vdash (A : P)M : Prop}.$$

Finally, we have term formation rules:

$$Abstr : \frac{\Gamma \vdash T : K \quad \Gamma[x:T] \vdash P : K' \quad \Gamma[x:T] \vdash M : P}{\Gamma \vdash [x:T]M : (x:T)P}$$

$$Appl : \frac{\Gamma \vdash M : (x:T)P \quad \Gamma \vdash N : T}{\Gamma \vdash (M\ N) : P\{N\}}.$$

**Remark.** The constant *Type* is a "type of all types". However, it is not itself of type *Type*.

**Definitions.** Let $\Gamma \vdash M : N$, with $\Gamma$ a valid context. When $N = Type$, we say that $M$ is a valid $\Gamma$-*type*. When $N = Prop$, we say that $M$ is a valid $\Gamma$-*proposition*. Finally, when $\Gamma \vdash M : N$, with $N$ a valid $\Gamma$-proposition, we say that $M$ is a $\Gamma$-*element*. The *pure* system of Constructions is obtained by deleting the third rule of context formation, which allows the introduction of *Type* variables. In the pure system, the only primitive type is *Prop*, and thus the only valid types are the products of the form $(A_1 : P_1)(A_2 : P_2) \cdots (A_n : P_n)Prop$.

We shall use a number of abbreviations. First, we write $\vdash M : P$ for $\{\} \vdash M : P$. Then, we give notations for the non-dependent products, that is for terms $(u : P)Q$ in the case where $u$ does not occur in $Q$. When both $P$ and $Q$ are propositions, we write $P \Rightarrow Q$. In other cases we write rather $P \rightarrow Q$. Finally, we abbreviate $(A : Prop)M$ into $\forall A \cdot M$ and $[A : Prop]M$ into $\Lambda A \cdot M$.

### 5.2.2 Adding type conversion

In the polymorphic $\lambda$-calculus seen in the last chapter, we defined propositional connectives as abbreviations. Thus for propositions $P$ and $Q$, the notation $P \wedge Q$ was just a meta-linguistic notation for the appropriate proposition. In the new calculus under consideration, connectives are indeed definable as expressions, and propositions are formed using the general rules of $\lambda$-calculus. We should therefore expect to need internal reduction rules for playing the rôle of macro-expansion.

It is indeed the case that such rules are necessary for type-checking. For instance, let us assume we define conjunction along the ideas of the previous chapter:

$$\wedge \ := \ [P : Prop][Q : Prop](R : Prop)(P \Rightarrow Q \Rightarrow R) \Rightarrow R.$$

Now if we try to define the first projection, in a context

$$\Gamma = [P : Prop][Q : Prop][x : (\wedge\ P\ Q)],$$

we shall be unable to form the term $(x\ P)$, unless we are able to recognize that the type $(\wedge\ P\ Q)$ is equal (by $\beta$-conversion) to $(R : Prop) \cdots$.

The above discussion shows that some amount of type equality rules must be provided in a higher-order calculus. To what extent such rules should be explicit (from the point of view of a user checking a derivation using inference rules) is unclear. For instance, we may profit from meta-theoretical results (confluence, strong normalization) and convert all types to normal form using $\lambda$-calculus reduction rules. Now type equality is just identity of such canonical forms. But there is an obvious drawback here: we may spend useless time converting to normal form some types which could be recognized as different immediately by inspection of their head normal form. Thus $[u : A][v : A](u\ \cdots)$ and $[u : A][v : A](v\ \cdots)$ need not be reduced any further. This problem is aggravated by the fact that the higher-order nature of the calculus makes it possible to have subparts of types which are elements. For instance, if $P$ is a predicate over a propositional type $A$, then for any element $p : A$ we may have to convert $p$ to $q$ in order to apply $x : (P\ p)$ as argument to a proof of some lemma of type $(P\ q) \Rightarrow \cdots$.

We now present various rules of conversion which may be used to axiomatize type equality $\equiv$. Various sub-calculi are obtainable by taking a subset of these rules, together with the rule of type conversion:

$$\mathit{Type\ Equality} : \frac{\Gamma \vdash M : P \quad \Gamma \vdash P \equiv Q}{\Gamma \vdash M : Q}$$

$$\mathit{Refl} : \frac{\Gamma \vdash M : N}{\Gamma \vdash M \equiv M}$$

$$\mathit{Sym} : \frac{\Gamma \vdash M \equiv N}{\Gamma \vdash N \equiv M}$$

$$\mathit{Trans} : \frac{\Gamma \vdash M \equiv N \quad \Gamma \vdash N \equiv P}{\Gamma \vdash M \equiv P}$$

$$\mathit{Abseq} : \frac{\Gamma \vdash P_1 \equiv P_2 \qquad \Gamma[x : P_1] \vdash M_1 \equiv M_2}{\Gamma \vdash [x : P_1]M_1 \equiv [x : P_2]M_2}$$

$$\mathit{Quanteq} : \frac{\Gamma \vdash P_1 \equiv P_2 \qquad \Gamma[x : P_1] \vdash M_1 \equiv M_2}{\Gamma \vdash (x : P_1)M_1 \equiv (x : P_2)M_2}$$

$$\mathit{Appleq} : \frac{\Gamma \vdash (M\ N) : P \qquad \Gamma \vdash M \equiv M_1 \qquad \Gamma \vdash N \equiv N_1}{\Gamma \vdash (M\ N) \equiv (M_1\ N_1)}$$

$$\mathit{Beta} : \frac{\Gamma[x : A] \vdash M : P \qquad \Gamma \vdash N : A}{\Gamma \vdash ([x : A]M\ N) \equiv M\{N\}}$$

$$\mathit{Eta} : \frac{\Gamma \vdash M : P}{\Gamma \vdash [x : A](M^{+}\ x) \equiv M}.$$

Various subsystems can now be discussed. First, the rule *Eta* may be omitted. Then the rule *Abseq* (corresponding to the $\xi$ rule of $\lambda$-calculus) may be deleted, yielding a weak conversion system corresponding to combinatory conversion.

Finally, when the conversions at the level of the elements (i.e. the terms of type a proposition) are omitted, we get the *restricted* calculus of constructions.

The calculus of constructions presented above was defined in Th. Coquand's thesis [36], who proved the main meta-theoretic properties. Variations on the basic calculus are presented in [40,42].

### 5.2.3 Example

We want to define the intersection of a class of classes on a given type $A$. A natural attempt is to take

$$\mathit{Inter} \ := \ [C : (A \rightarrow \mathit{Prop}) \rightarrow \mathit{Prop}]\,[x : A](P : A \rightarrow \mathit{Prop})(C\ P) \rightarrow (P\ x).$$

Let us place ourselves in the context

$$\Gamma = [C_0 : (A \rightarrow \mathit{Prop}) \rightarrow \mathit{Prop}]\,[P_0 : A \rightarrow \mathit{Prop}]\,[p_0 : (C_0\ P_0)].$$

We shall build a proof of the inclusion of the predicate $(\mathit{Inter}\ C_0)$ in the predicate $P_0$. Let us consider

$$\Delta = \Gamma\,[x : A]\,[h : (\mathit{Inter}\ C_0\ x)].$$

We want to build with $p_0$, $x$, $h$, $P_0$, $C_0$ a term of type $(P_0\ x)$.

Intuitively, $h$ which is of type $(\mathit{Inter}\ C_0\ x)$ is also (by logical conversion using the definition of *Inter*) of type $(P : A \rightarrow \mathit{Prop})(C_0\ P) \Rightarrow (P\ x)$, and thus we may construct the term $(h\ P_0\ p_0)$. Now, taking:

$$\mathit{Subset} := [P : A \rightarrow \mathit{Prop}]\,[Q : A \rightarrow \mathit{Prop}]\,(x : A)P(x) \Rightarrow Q(x)\ :\ (P : A \rightarrow \mathit{Prop})(Q : A \rightarrow \mathit{Prop})\mathit{Prop},$$

we get

$$\Gamma \vdash [x : A]\,[h : (\mathit{Inter}\ C_0\ x)](h\ P_0\ p_0) : (\mathit{Subset}\ (\mathit{Inter}\ C_0)\ P_0).$$

This example shows that the conversion of types rules are absolutely needed as soon as one wants to develop mathematical proofs (note that this example can be developed in the restricted calculus as well as in the full calculus). The need for conversion rules is equally emphasized in [121] and [166].

### 5.2.4 Consistency

**Definition.** A proposition $\vdash P : \mathit{Prop}$ is *inhabited* if, and only if there is an element term $M$ such that $\vdash M : P$.

**Consistency Theorem.** The Calculus of Constructions is consistent, in the sense that there exists a proposition which is not inhabited.

The intuitive meaning of this statement is that the calculus does not prove all its well-formed propositions. Indeed, the term $\perp\ :=\ \forall A \cdot A$ *is such a proposition.*

## 5.3 Examples of constructions

All the examples discussed in polymorphic λ-calculus can be developed without modification in this new calculus, which extends it in a natural way. Let us now show how quantifiers can be expressed in the calculus.

### 5.3.1 Universal Quantification

Universal quantification, or general product, is implicit from the notation:

$$\Pi\ :=\ \Lambda A \cdot [P : A \rightarrow \mathit{Prop}]\ (x : A)(P\ x).$$

Π-introduction, i.e. universal generalization, is proved by abstraction:

$$\begin{aligned} \mathit{Gen}\ :=\ & \Lambda A \cdot [P : A \rightarrow \mathit{Prop}]\Lambda B \cdot [f : (x : A)\ B \Rightarrow (P\ x)]\,[y : B]\,[x : A](f\ x\ y) \\ & : \forall A \cdot (P : A \rightarrow \mathit{Prop})\forall B \cdot ((x : A)\ B \Rightarrow (P\ x)) \Rightarrow (B \Rightarrow (\Pi\ A\ P)). \end{aligned}$$

Similarly, Π-elimination is proved by instantiation, i.e. application:

$$\begin{aligned} \mathit{Inst}\ :=\ & \Lambda A \cdot [P : A \rightarrow \mathit{Prop}]\,[x : A]\,[p : (\Pi\ A\ P)](p\ x) \\ & : \forall A \cdot (P : A \rightarrow \mathit{Prop})(x : A)(\Pi\ A\ P) \Rightarrow (P\ x). \end{aligned}$$

### 5.3.2 Existential Quantification

Existential quantification, or general sum, can be defined by a generalization of the binary sum:

$$\Sigma := \Lambda A \cdot [P : A \to \mathit{Prop}] \forall B \cdot ((x : A)\,(P\ x) \Rightarrow B) \Rightarrow B.$$

We leave if as exercise to the reader to prove existential introduction and elimination:

$$\mathit{Exist} := \forall A \cdot (P : A \to \mathit{Prop})(x : A)\,(P\ x) \Rightarrow (\Sigma\ A\ P)$$

$$\mathit{Witness} := \forall A \cdot (P : A \to \mathit{Prop})\,(\Sigma\ A\ P) \Rightarrow A.$$

Note that in a certain sense existential quantification is an abstraction mechanism: from $(\Sigma\ A\ P)$ it is possible to get some $a : A$ such that $(P\ a)$, but *not* the proof $p : (P\ a)$ that it indeed satisfies predicate $P$. Thus the existential quantification of the calculus of constructions is fundamentally different from the sum in Martin-Löf's calculus [122].

### 5.3.3 Equality

Leibniz' equality is definable in the calculus:

$$\mathit{Equal} := \Lambda A \cdot [x : A]\,[y : A](P : A \to \mathit{Prop})(P\ x) \Rightarrow (P\ y).$$

**Exercise.** Define the properties for a polymorphic relation to be reflexive, symmetric and transitive. Give the three proofs that *Equal* verifies these properties.

### 5.3.4 Tarski's theorem

Let us now present a simple example of a higher-order proof. The goal is to prove Tarski's theorem [186]:

**Tarski's Theorem.** A function monotonous over a complete partial ordering admits a fixpoint.

The first difficulty in formalizing Tarski's theorem is to give it in as abstract a setting as possible, in order to get the most direct proof. Let us try the following. Let $A$ be a set, $R$ a transitive relation over $A$ which is complete, in the sense that every subset of $A$ has a least upper bound. Let $f : A \to A$ be monotonously increasing. Then $f$ admits a fixpoint.

We must now formalize the notions of set, subset, and fixpoint. A simple attempt at axiomatizing sets consists in assuming some type $A$ given with an equality relation $=$, and to represent sets in the "universe" $A$ by their characteristic predicate, i.e. as elements of type $A \to \mathit{Prop}$. As for fixpoint, it turns out that all we need to require is that for some $X$ we have $(R\ (f\ X)\ X)$ and $(R\ X\ (f\ X))$. That is, the only property of equality that is needed here is the fact that $R$ is anti-symmetric.

We thus assume that we are in a context $\Gamma$, containing the following hypotheses:
$[A : \mathit{Type}]$
$[= : A \to A \to \mathit{Prop}]$
$[R : A \to A \to \mathit{Prop}]$
$[\mathit{Rtrans} : (x : A)(y : A)(z : A)(R\ x\ y) \Rightarrow (R\ y\ z) \Rightarrow (R\ x\ z)]$
$[\mathit{Rantisym} : (x : A)(y : A)(R\ x\ y) \Rightarrow (R\ y\ x) \Rightarrow (=\ x\ y)]$
$[\mathit{lim} : (A \to \mathit{Prop}) \to A]$
$[\mathit{Upperb} : (P : A \to \mathit{Prop})(y : A)(P\ y) \Rightarrow (R\ y\ (\mathit{lim}\ P))]$

$[\mathit{Least} : (P : A \rightarrow \mathit{Prop})(y : A)((z : A)(P\ z) \Rightarrow (R\ z\ y)) \Rightarrow (R\ (\mathit{lim}\ P)\ y)]$
$[f : A \rightarrow A]$
$[\mathit{Incr} : (x : A)(y : A)(R\ x\ y) \Rightarrow (R\ (f\ x)\ (f\ y))]$

Now we consider the predicate $Q$ defined as:

$$Q \ := \ [u : A](R\ u\ (f\ u))$$

(that is, $Q$ is the set of pre-fixpoints of $f$) and the element $X : A$ defined as:

$$X \ := \ (\mathit{lim}\ Q).$$

The first part of the proof consists in showing a proof of $(R\ X\ (f\ X))$ in context $\Gamma$. Let us first consider $\Delta = \Gamma[y : A][h : (Q\ y)]$, and terms $M = (\mathit{Upperb}\ Q\ y)$ and $N = (\mathit{Incr}\ y\ X)$. We get:
$\Delta \vdash M : (R\ y\ (f\ y)) \Rightarrow (R\ y\ X)$, and:
$\Delta \vdash N : (R\ y\ X) \Rightarrow (R\ (f\ y)\ (f\ X))$. Composing the two proofs we get:
$\Delta \vdash M;N : (R\ y\ (f\ y)) \Rightarrow (R\ (f\ y)\ (f\ X))$.
Thus, taking $p = (M;N\ h)$, we obtain:
$\Delta \vdash (\mathit{Rtrans}\ y\ (f\ y)\ (f\ X)\ h\ p) : (R\ y\ (f\ X))$.
Discharging the hypotheses $h$ and $y$, we get $T = [y : A][h : (Q\ y)](\mathit{Rtrans}\ y\ (f\ y)\ (f\ X)\ h\ p)$ such that:
$\Gamma \vdash T : \forall y \in Q \cdot (R\ y\ (f\ X))$.
The proof is completed by constructing $U = (\mathit{Least}\ Q\ (f\ X)\ T)$, since:
$\Gamma \vdash U : (R\ X\ (f\ X))$.

The second part of the proof is the converse. Taking $Z = (\mathit{Incr}\ X\ (f\ X)\ U)$, we get:
$\Gamma \vdash Z : (R\ (f\ X)\ (f\ (f\ X)))$
but since this last proposition converts to $(Q\ (f\ X))$, we get:
$\Gamma \vdash (\mathit{Upperb}\ Q\ (f\ X)\ Z) : (R\ (f\ X)\ X)$.

The proof of Tarski's theorem is thus obtained as:
$\Gamma \vdash (\mathit{Rantisym}\ (f\ X)\ X\ (\mathit{Upperb}\ Q\ (f\ X)\ Z)\ U) : (=\ (f\ X)\ X)$.

**Exercise.** Use the above argument and the quantifier manipulation combinators above to prove Tarski's theorem as a fully quantified statement.

Numerous examples of proofs verified on machine are presented in [39]. A general discussion on the formalization of mathematical arguments in higher order intuitionistic logic is given in [164].

## 6 A constructive theory of types

Let us now augment the Calculus of Constructions with rules allowing for the abstraction over all types. The first natural attempt is to allow $\mathit{Type} : \mathit{Type}$. We would thus get a system of rules very close to the one considered by P. Martin-Löf in [118]. However, this was shown to be inconsistent by Girard, who showed that it was possible to encode the paradox of Burali-Forti in such a system. An abstract analysis of such paradoxes is given by Coquand in [37]. Coquand showed that it was possible to quantify propositions over all types, but *not* other types such as product types. Such a system is presented below.

## 6.1 A system for uniform proofs

First, two rules provide for abstraction over all types:

$$TypeQuant : \frac{\Gamma[t : Type] \vdash P : Prop}{\Gamma \vdash (t : Type)P : Prop}$$

$$TypeAbstr : \frac{\Gamma[t : Type] \vdash P : Prop \quad \Gamma[t : Type] \vdash M : P}{\Gamma \vdash [t : Type]M : (t : Type)P}.$$

Finally, we give one more type conversion rule:

$$TypeEq : \frac{\Gamma[t : Type] \vdash P : Prop \qquad \Gamma[t : Type] \vdash P \equiv Q}{\Gamma \vdash (t : Type)P \equiv (t : Type)Q}.$$

In such a system, we may now abstract the above proof of Tarski's theorem.

## 6.2 A system with a hierarchy of universes

It is even possible to iterate the idea of a type gathering all the types obtained so far. One thus gets a system with a hierarchy of universes like in Martin-Löf's system [122]. Let us present along those lines Coquand's Generalized Calculus of Constructions [37].

### 6.2.1 Terms

1. $Type(i)$, for $i$ non-negative integer, and $Prop$ are terms
2. a variable $x$ is a term
3. if $M$ and $N$ are terms, then $(M\ N)$ is a term (application)
4. if $M$ and $N$ are terms, then $[x : M]N$ is a term (abstraction)
5. if $M$ and $N$ are terms, then $(x : M)N$ is a term (product). As previously, we denote by $\equiv$ the relation of $\lambda\beta$-conversion between terms.

### 6.2.2 Contexts

Contexts are ordered lists of *bindings* of the form $x : M$, where $x$ is a variable and $M$ is a term. Not every context is valid. The following rules define the valid contexts.

the empty context is valid

$$\frac{\Gamma \text{ is valid} \quad \Gamma \vdash M : Prop \quad x \text{ is not bound in } \Gamma}{\Gamma, x : M \text{ is valid}}$$

$$\frac{\Gamma \text{ is valid} \quad \Gamma \vdash M : Type(i) \quad x \text{ is not bound in } \Gamma}{\Gamma, x : M \text{ is valid}}$$

These rules are defined mutually recursively with the following type inference rules, which define the judgements $\Gamma \vdash M : N$, to be read "the term $M$ is of type $N$ in context $\Gamma$".

### 6.2.3 Type Inference Rules

$$\frac{\Gamma \text{ is valid}}{\Gamma \vdash \mathit{Prop} : \mathit{Type}(0)}$$

$$\frac{\Gamma \text{ is valid}}{\Gamma \vdash \mathit{Type}(i) : \mathit{Type}(i+1)} \qquad (*)$$

$$\frac{\Gamma \vdash M : \mathit{Type}(i)}{\Gamma \vdash M : \mathit{Type}(i+1)} \qquad \text{(coerce)}$$

$$\frac{\Gamma \text{ is valid} \quad x : M \in \Gamma}{\Gamma \vdash x : M}$$

$$\frac{\Gamma, x : M \vdash N : P}{\Gamma \vdash [x : M]N : (x : M)P}$$

$$\frac{\Gamma, x : M \vdash N : \mathit{Prop}}{\Gamma \vdash (x : M)N : \mathit{Prop}}$$

$$\frac{\Gamma \vdash M : \mathit{Type}(j) \quad \Gamma, x : M \vdash N : \mathit{Type}(i)}{\Gamma \vdash (x : M)N : \mathit{Type}(\mathit{max}(i,j))}$$

$$\frac{\Gamma \vdash M : \mathit{Prop} \quad \Gamma, x : M \vdash N : \mathit{Type}(i)}{\Gamma \vdash (x : M)N : \mathit{Type}(i)} \qquad (**)$$

$$\frac{\Gamma \vdash M : (x : Q)P \quad \Gamma \vdash N : R \quad Q \equiv R}{\Gamma \vdash (M\ N) : [N/x]P}$$

The only serious departure from [37] is the addition of rule $(*)$, which was inadvertently ommitted, and of rule $(**)$, which is needed to prove the following lemma.

**Lemma.** If $\Gamma \vdash M : N$ is derivable, then either $\Gamma \vdash N : \mathit{Prop}$ is derivable, in which case we say that $M$ is a *proof* of *proposition* $N$ in context $\Gamma$, or else $\Gamma \vdash N : \mathit{Type}(i)$ is derivable for some $i \geq 0$, in which case we say that $M$ is a *realization* of *specification* $N$ in context $\Gamma$.

This lemma shows that there are two distinct kinds of types in the system, in the sense of terms appearing to the right of a colon in a derivable sequent.

### 6.2.4 A digression on types, specifications and propositions

We say that term $T$ is a *type* (in a given context) if it is either a specification or a proposition. We remark that the rules for context formation are that variables may be bound only to types, not to arbitrary terms. Since these are the two kinds of bindings, we shall speak of the constants *Prop* and *Type*$(i)$ of the system as the *kinds*, following the MacQueen-Sethi terminology[112]. Specifications are the natural generalization of the notion of types in the sense of Church's theory of types. They are more general in that the product formation operator is *dependent*, like in Martin-Löf's theory of types [122]. When $x$ does not occur in $N$, the specification $(x : M)N$ may be abbreviated in the more traditional $M \rightarrow N$. For instance, the specification of a predicate over type $T$ would be $T \rightarrow \mathit{Prop}$. Similarly, when $P$ is a proposition and $Q$ is a proposition in which $x$ does not occur, we may abbreviate $(x : P)Q$ in $P \Rightarrow Q$. Also, we use $\forall x : M \cdot P$ for $(x : M)P$ when $M$ is a specification and $P$ is a proposition. When $P$ is a proposition and $M$ is a specification, the specification

$(x : P)M$ has realizations depending on the proof of $P$. It is not usual to consider such types in ordinary logic. However, they are needed to formalize constructive mathematics in Bishop's sense, where evidence of properties is taken as computationally meaningful. Here evidence (of properties) is internalized as proofs (of propositions). This is in contrast to the formalism LF (logical framework) developed at the University of Edinburgh [69], where judgements (as opposed to propositions) are types. We refer to [123] for a philosophical discussion of the issues involved.

Remark that the only specifications $P$ which are typable of type $Type(0)$ in the empty context are (convertible to) the terms of the form:

$$(x_1 : M_1)(x_1 : M_1)...(x_1 : M_1)Prop.$$

The types of the system are more general than just specifications, since we use the paradigm of propositions as types [72]. More precisely, the formulation of the logical part of the system in natural deduction style allows the use of $\lambda$-abstraction for the dual purpose of building functional realizations as well as building proofs under hypotheses.

The inference system is completed by type equality rules, as follows.

### 6.2.5 Type Equality Rules

$$\frac{\Gamma \vdash M : N \quad \Gamma \vdash P : Prop \quad N \equiv P}{\Gamma \vdash M : P}$$

$$\frac{\Gamma \vdash M : N \quad \Gamma \vdash P : Type(i) \quad N \equiv P}{\Gamma \vdash M : P}.$$

Note that we allow $\lambda$-conversion only for types, not for other terms.

**Remark 1.** It might seem that the previous lemma allows to *simplify the two rules* in one simpler rule:

$$\frac{\Gamma \vdash M : N \quad N \equiv P}{\Gamma \vdash M : P}.$$

However, we are careful to specify that $P$ must be itself well-typed, since otherwise we might introduce non-typable terms as types of other terms. Indeed, we need this restriction in order to preserve the validity of the lemma above.

**Remark 2.** The types equality rules allow us to replace the rule of application by the simpler:

$$\frac{\Gamma \vdash M : (x : Q)P \quad \Gamma \vdash N : Q}{\Gamma \vdash (M\ N) : [N/x]P}$$

Indeed, this is the way it was formulated originally [37]. However, our formulation is more consistent from the point of view of the meaning of the meta-variables in the rules, since several occurrences of the same meta-variable should mean that the corresponding term or context is *shared*, and this is not the case for $Q$ above.

The system GCC is quite powerful. It extends strictly Girard's higher order system $F^\omega$. It permits to formalize completely the Principia's, including the so-called "typical ambiguity" feature. However, it is not very convenient to use, since we have to explicitly manipulate the universe hierarchy. Furthermore, there is no unicity of types (even modulo lambda-conversion), because of rule (*coerce*). This difficulty may be solved by manipulating the integer arguments to the *Type* constant as symbolic expressions. This is explained in [83].

## References

[1] A. Aho, J. Hopcroft, J. Ullman. "The Design and Analysis of Computer Algorithms." Addison-Wesley (1974).

[2] P. B. Andrews. "Resolution in Type Theory." Journal of Symbolic Logic **36,3** (1971), 414–432.

[3] P. B. Andrews, D. A. Miller, E. L. Cohen, F. Pfenning. "Automating higher-order logic." Dept of Math, University Carnegie-Mellon, (Jan. 1983).

[4] H. Barendregt. "The Lambda-Calculus: Its Syntax and Semantics." North-Holland (1980).

[5] H. Barendregt and A. Rezus. "Semantics for Classical AUTOMATH and Related Systems." Information and Control **59** (1983) 127–147.

[6] E. Bishop. "Foundations of Constructive Analysis." McGraw-Hill, New-York (1967).

[7] E. Bishop. "Mathematics as a numerical language." Intuitionism and Proof Theory, Eds. J. Myhill, A.Kino and R.E.Vesley, North-Holland, Amsterdam, (1970) 53–71.

[8] C. Böhm, A. Berarducci. "Automatic Synthesis of Typed Lambda-Programs on Term Algebras." Unpublished manuscript, (June 1984).

[9] R.S. Boyer, J Moore. "The sharing of structure in theorem proving programs." Machine Intelligence **7** (1972) Edinburgh U. Press, 101–116.

[10] R. Boyer, J Moore. "A Lemma Driven Automatic Theorem Prover for Recursive Function Theory." 5th International Joint Conference on Artificial Intelligence, (1977) 511–519.

[11] R. Boyer, J Moore. "A Computational Logic." Academic Press (1979).

[12] R. Boyer, J Moore. "A mechanical proof of the unsolvability of the halting problem." Report ICSCA-CMP-28, Institute for Computing Science, University of Texas at Austin (July 1982).

[13] R. Boyer, J Moore. "Proof Checking the RSA Public Key Encryption Algorithm." Report ICSCA-CMP-33, Institute for Computing Science, University of Texas at Austin (Sept. 1982).

[14] R. Boyer, J Moore. "Proof checking theorem proving and program verification." Report ICSCA-CMP-35, Institute for Computing Science, University of Texas at Austin (Jan. 1983).

[15] N.G. de Bruijn. "The mathematical language AUTOMATH, its usage and some of its extensions." Symposium on Automatic Demonstration, IRIA, Versailles, 1968. Printed as Springer-Verlag Lecture Notes in Mathematics **125**, (1970) 29–61.

[16] N.G. de Bruijn. "Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem." Indag. Math. **34,5** (1972), 381–392.

[17] N.G. de Bruijn. "Automath a language for mathematics." Les Presses de l'Université de Montréal, (1973).

[18] N.G. de Bruijn. "Some extensions of Automath: the AUT-4 family." Internal Automath memo M10 (Jan. 1974).

[19] N.G. de Bruijn. "A survey of the project Automath." (1980) in to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Eds Seldin J. P. and Hindley J. R., Academic Press (1980).

[20] M. Bruynooghe. "The Memory Management of PROLOG implementations." Logic Programming Workshop. Ed. Tarnlund S.A (July 1980).

[21] R. Burstall and B. Lampson. "A Kernel Language for Modules and Abstract Data Types." International Symposium on Semantics of Data Types, Sophia-Antipolis. Proceedings Eds. G. Kahn, D. B. MacQueen and G. Plotkin, Springer-Verlag LNCS 173, 1-50.

[22] L. Cardelli. "ML under UNIX." Bell Laboratories, Murray Hill, New Jersey (1982).

[23] L. Cardelli. "Amber." Bell Laboratories Technical Memorandum TM 11271-840924-10 (1984).

[24] L. Cardelli. "The Amber Machine." Bell Laboratories, Murray Hill, New Jersey (1985).

[25] L. Cardelli. "Basic Polymorphism Type-checking." Polymorphism, Jan. 1985

[26] L. Cardelli. "A Polymorphic $\lambda$-calculus with Type:Type." DEC SRC Report 10 (May 1986).

[27] L. Cardelli and P. Wegner. "On Understanding Types, Data abstraction, and Polymorphism." ACM Computing Surveys **17** (Dec. 1985) 471-522.

[28] D. de Champeaux. "About the Paterson-Wegman Linear Unification Algorithm." J. of Comp. and System Sciences **32** (1986) 79-90.

[29] A. Church. "A formulation of the simple theory of types." Journal of Symbolic Logic **5**,1 (1940) 56-68.

[30] A. Church. "The Calculi of Lambda-Conversion." Princeton U. Press, Princeton N.J. (1941).

[31] A. Colmerauer, H. Kanoui, R. Pasero, Ph. Roussel. "Un système de communication homme-machine en francais." Rapport de recherche, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille (1973).

[32] R.L. Constable, J.L. Bates. "Proofs as Programs." Dept. of Computer Science, Cornell University. (Feb. 1983).

[33] R.L. Constable, J.L. Bates. "The Nearly Ultimate Pearl." Dept. of Computer Science, Cornell University. (Dec. 1983).

[34] R.L. Constable, N.P. Mendler. "Recursive Definitions in Type Theory." Private Communication (1985).

[35] R.L. Constable et al. "Implementing Mathematics in the NuPrl System." Prentice-Hall (1986).

[36] Th. Coquand. "Une théorie des constructions." Thèse de troisième cycle, Université Paris VII (Jan. 85).

[37] Th. Coquand. "An analysis of Girard's paradox." First Conference on Logic in Computer Science, Boston (June 1986).

[38] Th. Coquand, G. Huet. "A Theory of Constructions." Preliminary version, presented at the International Symposium on Semantics of Data Types, Sophia-Antipolis (June 84).

[39] Th. Coquand, G. Huet. "Constructions: A Higher Order Proof System for Mechanizing Mathematics." EUROCAL85, Linz, Springer-Verlag LNCS 203 (1985).

[40] Th. Coquand, G. Huet. "Concepts Mathématiques et Informatiques Formalisés dans le Calcul des Constructions." Colloque de Logique, Orsay (Juil. 1985).

[41] Th. Coquand, G. Huet. "A Selected Bibliography on Constructive Mathematics, Intuitionistic Type Theory and Higher Order Deduction." J. Symbolic Computation (1985) 1 323–328.

[42] Th. Coquand, G. Huet. "The Calculus of Constructions." To appear, Information and Control (1986).

[43] J. Corbin, M. Bidoit. "A Rehabilitation of Robinson's Unification Algorithm." IFIP 83, Elsevier Science (1983) 909–914.

[44] G. Cousineau, P.L. Curien and M. Mauny. "The Categorical Abstract Machine." In Functional Programming Languages and Computer Architecture, Ed. J. P. Jouannaud, Springer-Verlag LNCS 201 (1985) 50–64.

[45] P.L. Curien. "Combinateurs catégoriques, algorithmes séquentiels et programmation applicative." Thèse de Doctorat d'Etat, Université Paris VII (Dec. 1983).

[46] P. L. Curien. "Categorical Combinatory Logic." ICALP 85, Nafplion, Springer-Verlag LNCS 194 (1985).

[47] P.L. Curien. "Categorical Combinators, Sequential Algorithms and Functional Programming." Pitman (1986).

[48] H. B. Curry, R. Feys. "Combinatory Logic Vol. I." North-Holland, Amsterdam (1958).

[49] D. Van Daalen. "The language theory of Automath." Ph. D. Dissertation, Technological Univ. Eindhoven (1980).

[50] Luis Damas, Robin Milner. "Principal type-schemas for functional programs." Edinburgh University (1982).

[51] A. Demers and J. Donahue. "Datatypes, parameters and type checking." 7th ACM Symposium on Principles of Programming Languages, Las Vegas (1980), 12-23.

[52] P.J. Downey, R. Sethi, R. Tarjan. "Variations on the common subexpression problem." JACM **27,4** (1980) 758–771.

[53] M. Dummett. "Elements of Intuitionism." Clarendon Press, Oxford (1977).

[54] F. Fages. "Formes canoniques dans les algèbres booléennes et application à la démonstration automatique en logique de premier ordre." Thèse de 3ème cycle, Univ. de Paris VI (Juin 1983).

[55] F. Fages. "Associative-Commutative Unification." Submitted for publication (1985).

[56] F. Fages, G. Huet. "Unification and Matching in Equational Theories." CAAP 83, l'Aquila, Italy. In Springer-Verlag LNCS **159** (1983).

[57] P. Flajolet, J.M. Steyaert. "On the Analysis of Tree-Matching Algorithms." in Automata, Languages and Programming 7th Int. Coll., Lecture Notes in Computer Science **85** Springer Verlag (1980) 208–219.

[58] S. Fortune, D. Leivant, M. O'Donnell. "The Expressiveness of Simple and Second Order Type Structures." Journal of the Assoc. for Comp. Mach., **30,1**, (Jan. 1983) 151–185.

[59] G. Frege. "Begriffschrift, a formula language, modeled upon that of arithmetic, for pure thought." (1879). Reprinted in From Frege to Gödel, J. van Heijenoort, Harvard University Press, 1967.

[60] G. Gentzen. "The Collected Papers of Gerhard Gentzen." Ed. E. Szabo, North-Holland, Amsterdam (1969).

[61] J.Y. Girard. "Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. Proceedings of the Second Scandinavian Logic Symposium, Ed. J.E. Fenstad, North Holland (1970) 63–92.

[62] J.Y. Girard. "Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieure." Thèse d'Etat, Université Paris VII (1972).

[63] K. Gödel. "Uber eine bisher noch nicht benutze Erweitrung des finiten Standpunktes." Dialectica, **12** (1958).

[64] W. D. Goldfarb. "The Undecidability of the Second-order Unification Problem." Theoretical Computer Science, **13**, (1981) 225–230.

[65] M. Gordon, R. Milner, C. Wadsworth. "A Metalanguage for Interactive Proof in LCF." Internal Report CSR-16-77, Department of Computer Science, University of Edinburgh (Sept. 1977).

[66] M. J. Gordon, A. J. Milner, C. P. Wadsworth. "Edinburgh LCF" Springer-Verlag LNCS **78** (1979).

[67] W. E. Gould. "A Matching Procedure for Omega Order Logic." Scientific Report 1, AFCRL 66-781, contract AF19 (628)-3250 (1966).

[68] J. Guard. "Automated Logic for Semi-Automated Mathematics." Scientific Report 1, AFCRL (1964).

[69] R. Harper, F. Honsell and G. Plotkin. "The Edinburgh Logical Framework." Private communication (Oct. 1986).

[70] J. Herbrand. "Recherches sur la théorie de la démonstration." Thèse, U. de Paris (1930). In: Ecrits logiques de Jacques Herbrand, PUF Paris (1968).

[71] C. M. Hoffmann, M. J. O'Donnell. "Programming with Equations." ACM Transactions on Programming Languages and Systems, **4,1** (1982) 83–112.

[72] W. A. Howard. "The formulæ-as-types notion of construction." Unpublished manuscript (1969). Reprinted in to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Eds Seldin J. P. and Hindley J. R., Academic Press (1980).

[73] G. Huet. "Constrained Resolution: a Complete Method for Type Theory." Ph.D. Thesis, Jennings Computing Center Report 1117, Case Western Reserve University (1972).

[74] G. Huet. "A Mechanization of Type Theory." Proceedings, 3rd IJCAI, Stanford (Aug. 1973).

[75] G. Huet. "The Undecidability of Unification in Third Order Logic." Information and Control **22** (1973) 257–267.

[76] G. Huet. "A Unification Algorithm for Typed Lambda Calculus." Theoretical Computer Science, **1.1** (1975) 27–57.

[77] G. Huet. "Résolution d'équations dans des langages d'ordre 1,2, ... $\omega$." Thèse d'Etat, Université Paris VII (1976).

[78] G. Huet. "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems." J. Assoc. Comp. Mach. **27,4** (1980) 797–821.

[79] G. Huet. "A Complete Proof of Correctness of the Knuth-Bendix Completion Algorithm." JCSS **23,1** (1981) 11-21.

[80] G. Huet. "Initiation à la Théorie des Catégories." Polycopié de cours de DEA, Université Paris VII (Nov. 1985).

[81] G. Huet. "Cartesian Closed Categories and Lambda-Calculus." Category Theory Seminar, Carnegie-Mellon University (Dec. 1985).

[82] G. Huet. "Formal Structures for Computation and Deduction." Course Notes, Carnegie-Mellon University, May 1986.

[83] G. Huet. "Extending the Calculus of Constructions with Type:Type." In preparation.

[84] G. Huet, J.M. Hullot. "Proofs by Induction in Equational Theories With Constructors." JCSS **25,2** (1982) 239–266.

[85] G. Huet, J.J. Lévy "Call by Need Computations in Non-Ambiguous Linear Term Rewriting Systems." Rapport Laboria 359, IRIA (Aug. 1979).

[86] G. Huet, D. Oppen. "Equations and Rewrite Rules: a Survey." In Formal Languages: Perspectives and Open Problems, Ed. Book R., Academic Press (1980).

[87] L.S. Jutting. "A translation of Landau's "Grundlagen" in AUTOMATH." Eindhoven University of Technology, Dept of Mathematics (Oct. 1976).

[88] L.S. van Benthem Jutting. "The language theory of $\Lambda_\infty$, a typed $\lambda$-calculus where terms are types." Unpublished manuscript (1984).

[89] G. Kahn, G. Plotkin. "Domaines concrets." Rapport Laboria 336, IRIA (Déc. 1978).

[90] D. Clément, J. Despeyroux, T. Despeyroux, G. Kahn. "Natural semantics on the computer." Research Report 416, Inria, June 1985.

[91] D. Clément, J. Despeyroux, T. Despeyroux, G. Kahn. "A Simple Applicative Language: mini-ML." Research Report to appear, Inria, 1986.

[92] P. Kanellakis, J. Mitchell. "Polymorphic unification and ML typing." Proceedings, 16th Annual ACM Symposium on Principles of Programming Languages (Jan. 1989), pp. 105–113.

[93] J. Ketonen, J. S. Weening. "The language of an interactive proof checker." Stanford University (1984).

[94] J. Ketonen. "EKL-A Mathematically Oriented Proof Checker." 7th International Conference on Automated Deduction, Napa, California (May 1984). Springer-Verlag LNCS **170**.

[95] J. Ketonen. "A mechanical proof of Ramsey theorem." Stanford Univ. (1983).

[96] S.C. Kleene. "On the interpretation of intuitionistic number theory." J. Symbolic Logic **10** (1945).

[97] S.C. Kleene. "Introduction to Meta-mathematics." North Holland (1952).

[98] J.W. Klop. "Combinatory Reduction Systems." Ph. D. Thesis, Mathematisch Centrum Amsterdam (1980).

[99] D.E. Knuth, J. Morris, V. Pratt. "Fast Pattern Matching in Strings." SIAM Journal on Computing **6,2** (1977) 323–350.

[100] G. Kreisel. "On the interpretation of nonfinitist proofs, Part I, II." JSL **16,17** (1952, 1953).

[101] J. Lambek. "From Lambda-calculus to Cartesian Closed Categories." in To H. B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism, Eds. J. P. Seldin and J. R. Hindley, Academic Press (1980).

[102] J. Lambek and P. J. Scott. "Aspects of Higher Order Categorical Logic." Contemporary Mathematics **30** (1984) 145–174.

[103] J. Lambek and P. J. Scott. "Introduction to higher order categorical logic." Cambridge University Press (1986).

[104] P. J. Landin. "The next 700 programming languages." Comm. ACM **9,3** (1966) 157–166.

[105] Philippe Le Chenadec. "Formes canoniques dans les algèbres finiment présentées." Thèse de 3ème cycle, Univ. d'Orsay (Juin 1983).

[106] D. Leivant. "Polymorphic type inference." 10th ACM Symposium on Principles of Programming Languages (1983).

[107] D. Leivant. "Structural semantics for polymorphic data types." 10th ACM Conference on Principles of Programming Languages (1983).

[108] J.J. Lévy. "Réductions correctes et optimales dans le λ-calcul." Thèse d'Etat, U. Paris VII (1978).

[109] J.J. Lévy. "Optimal Reductions in the λ-calculus." in To H. B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism, Eds. J. P. Seldin and J. R. Hindley, Academic Press (1980).

[110] S. MacLane. "Categories for the Working Mathematician." Springer-Verlag (1971).

[111] D. MacQueen, G. Plotkin, R. Sethi. "An ideal model for recursive polymorphic types." Proceedings, Principles of Programming Languages Symposium, Jan. 1984, 165–174.

[112] D. B. MacQueen, R. Sethi. "A semantic model of types for applicative languages." ACM Symposium on Lisp and Functional Programming (Aug. 1982).

[113] E.G. Manes. "Algebraic Theories." Springer-Verlag (1976).

[114] C. Mann. "The Connection between Equivalence of Proofs and Cartesian Closed Categories." Proc. London Math. Soc. **31** (1975) 289–310.

[115] A. Martelli, U. Montanari. "Theorem proving with structure sharing and efficient unification." Proc. 5th IJCAI, Boston, (1977) p 543.

[116] A. Martelli, U. Montanari. "An Efficient Unification Algorithm." ACM Trans. on Prog. Lang. and Syst. **4,2** (1982) 258–282.

[117] William A. Martin. "Determining the equivalence of algebraic expressions by hash coding." JACM **18,4** (1971) 549–558.

[118] P. Martin-Löf. "A theory of types." Report 71-3, Dept. of Mathematics, University of Stockholm, Feb. 1971, revised (Oct. 1971).

[119] P. Martin-Löf. "About models for intuitionistic type theories and the notion of definitional equality." Paper read at the Orléans Logic Conference (1972).

[120] P. Martin-Löf. "An intuitionistic Theory of Types: predicative part." Logic Colloquium 73, Eds. H. Rose and J. Sepherdson, North-Holland, (1974) 73–118.

[121] P. Martin-Löf. "Constructive Mathematics and Computer Programming." In Logic, Methodology and Philosophy of Science **6** (1980) 153–175, North-Holland.

[122] P. Martin-Löf. "Intuitionistic Type Theory." Studies in Proof Theory, Bibliopolis (1984).

[123] P. Martin-Löf. "Truth of a proposition, evidence of a judgement, validity of a proof." Transcript of talk at the workshop "Theories of Meaning", Centro Fiorentino di Storia e Filosofia della Scienza, Villa di Mondeggi, Florence (June 1985).

[124] J. Mc Carthy. "Recursive functions of symbolic expressions and their computation by machine." CACM **3,4** (1960) 184–195.

[125] N. McCracken. "An investigation of a programming language with a polymorphic type structure." Ph.D. Dissertation, Syracuse University (1979).

[126] A. R. Meyer and M. B. Reinhold. "Type is Not a Type: Preliminary Report." Proceedings, Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida (Jan. 1986) 287–295.

[127] D.A. Miller. "Proofs in Higher-order Logic." Ph. D. Dissertation, Carnegie-Mellon University (Aug. 1983).

[128] D.A. Miller. "Expansion tree proofs and their conversion to natural deduction proofs." Technical report MS-CIS-84-6, University of Pennsylvania (Feb. 1984).

[129] R. Milner. "A Theory of Type Polymorphism in Programming." Journal of Computer and System Sciences **17** (1978) 348–375.

[130] R. Milner. "A proposal for Standard ML." Report CSR-157-83, Computer Science Dept., University of Edinburgh (1983).

[131] C. Mohring. "Algorithm Development in the Calculus of Constructions." IEEE Symposium on Logic in Computer Science, Cambridge, Mass. (June 1986).

[132] A. Mycroft. "Polymorphic Type Schemes and Recursive Definitions". 6th International Symposium on Programming, Springer-Verlag LNCS **167** (1984) 217–228.

[133] R.P. Nederpelt. "Strong normalization in a typed $\lambda$ calculus with $\lambda$ structured types." Ph. D. Thesis, Eindhoven University of Technology (1973).

[134] R.P. Nederpelt. "An approach to theorem proving on the basis of a typed $\lambda$-calculus." 5th Conference on Automated Deduction, Les Arcs, France. Springer-Verlag LNCS **87** (1980).

[135] G. Nelson, D.C. Oppen. "Fast decision procedures based on congruence closure." JACM **27,2** (1980) 356–364.

[136] M.H.A. Newman. "On Theories with a Combinatorial Definition of "Equivalence"." Annals of Math. **43,2** (1942) 223–243.

[137] B. Nordström. "Programming in Constructive Set Theory: Some Examples." Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, Portmouth, New Hampshire (Oct. 1981) 141–154.

[138] B. Nordström. "Description of a Simple Programming Language." Report 1, Programming Methodology Group, University of Goteborg (Apr. 1984).

[139] B. Nordström, K. Petersson. "Types and Specifications." Information Processing 83. Ed. R. Mason, North-Holland, (1983) 915–920.

[140] B. Nordström, J. Smith. "Propositions and Specifications of Programs in Martin-Löf's Type Theory." BIT **24**, (1984) 288–301.

[141] A. Obtulowicz. "The Logic of Categories of Partial Functions and its Applications." Dissertationes Mathematicae 241 (1982).

[142] M.S. Paterson, M.N. Wegman. "Linear Unification." J. of Computer and Systems Sciences **16** (1978) 158–167.

[143] L. Paulson. "Recent Developments in LCF : Examples of structural induction." Technical Report No 34, Computer Laboratory, University of Cambridge (Jan. 1983).

[144] L. Paulson. "Tactics and Tacticals in Cambridge LCF." Technical Report No 39, Computer Laboratory, University of Cambridge (July 1983).

[145] L. Paulson. "Verifying the unification algorithm in LCF." Technical report No 50, Computer Laboratory, University of Cambridge (March 1984).

[146] L. C. Paulson. "Constructing Recursion Operators in Intuitionistic Type Theory." Tech. Report 57, Computer Laboratory, University of Cambridge (Oct. 1984).

[147] G.E. Peterson, M.E. Stickel. "Complete Sets of Reduction for Equational Theories with Complete Unification Algorithms." JACM **28,2** (1981) 233–264.

[148] T. Pietrzykowski, D.C. Jensen. "A complete mechanization of $\omega$-order type theory." Proceedings of ACM Annual Conference (1972).

[149] T. Pietrzykowski. "A Complete Mechanization of Second-Order Type Theory." JACM **20** (1973) 333–364.

[150] D. Prawitz. "Natural Deduction." Almqist and Wiskell, Stockolm (1965).

[151] D. Prawitz. "Ideas and results in proof theory." Proceedings of the Second Scandinavian Logic Symposium (1971).

[152] W. V. Quine. "The problem of simplifying truth functions." Amer. Math. Monthly bf 59,8 (1952) 521–531.

[153] H. Rasiowa, R. Sikorski "The Mathematics of Metamathematics." Monografie Matematyczne tom **41**, PWN, Polish Scientific Publishers, Warszawa (1963).

[154] J. C. Reynolds. "Definitional Interpreters for Higher Order Programming Languages." Proc. ACM National Conference, Boston, (Aug. 72) 717–740.

[155] J. C. Reynolds. "Towards a Theory of Type Structure." Programming Symposium, Paris. Springer Verlag LNCS **19** (1974) 408–425.

[156] J. C. Reynolds. "Types, abstraction, and parametric polymorphism." IFIP Congress'83, Paris (Sept. 1983).

[157] J. C. Reynolds. "Polymorphism is not set-theoretic." International Symposium on Semantics of Data Types, Sophia-Antipolis (June 1984).

[158] J. C. Reynolds. "Three approaches to type structure." TAPSOFT Advanced Seminar on the Role of Semantics in Software Development, Berlin (March 1985).

[159] J. A. Robinson. "A Machine-Oriented Logic Based on the Resolution Principle." JACM **12** (1965) 32–41.

[160] J. A. Robinson. "Computational Logic: the Unification Computation." Machine Intelligence **6** Eds B. Meltzer and D.Michie, American Elsevier, New-York (1971).

[161] B. Russel and A.N. Whitehead. "Principia Mathematica." Volume 1,2,3 Cambridge University Press (1912).

[162] K. Schütte. "Proof theory." Springer-Verlag (1977).

[163] D. Scott. "Constructive validity." Symposium on Automatic Demonstration, Springer-Verlag Lecture Notes in Mathematics, **125** (1970).

[164] D. Scott. "Identity and existence in intuitionistic logic." Proceedings of the Research Symposium on Applications of Sheaf Theory to Logic, Algebra and Analysis, Durham (July 1977), Eds. M.P. Fourman, C.J. Mulvey and D.S. Scott. Lecture Notes in Mathematics 753, Springer-Verlag (1979) 660–696.

[165] D. Scott. "Relating Theories of the Lambda-Calculus." in To H. B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism, Eds. J. P. Seldin and J. R. Hindley, Academic Press (1980).

[166] J.P. Seldin. "Progress report on generalized functionality." Ann. Math. Logic. 17 (1979).

[167] J.R. Shoenfield. "Mathematical Logic." Addison-Wesley (1967).

[168] R.E. Shostak "Deciding Combinations of Theories." JACM **31,1** (1985) 1–12.

[169] J. Smith. "Course-of-values recursion on lists in intuitionistic type theory." Unpublished notes, Göteborg University (Sept. 1981).

[170] J. Smith. "The identification of propositions and types in Martin-Lof's type theory : a programming example." International Conference on Foundations of Computation Theory, Borgholm, Sweden, (Aug. 1983) Springer-Verlag LNCS **158**.

[171] R. Statman. "Intuitionistic Propositional Logic is Polynomial-space Complete." Theoretical Computer Science **9** (1979) 67–72, North-Holland.

[172] R. Statman. "The typed Lambda-Calculus is not Elementary Recursive." Theoretical Computer Science **9** (1979) 73–81.

[173] R. Statman. "Number theoretic functions computable by polymorphic programs." 22nd IEE Symp. on Foundations of Computer Science (1981) 279–282.

[174] R. Statman. "On translating λ-terms into combinators; the basis problem." IEEE Conference on Logic in Computer Science, Boston, June 1986.

[175] S. Stenlund. "Combinators λ-terms, and proof theory." Reidel (1972).

[176] M.E. Stickel "A Complete Unification Algorithm for Associative-Commutative Functions." JACM **28,3** (1981) 423–434.

[177] M.E. Szabo. "Algebra of Proofs." North-Holland (1978).

[178] W. Tait. "A non constructive proof of Gentzen's Hauptsatz for second order predicate logic." Bull. Amer. Math. Soc. **72** (1966).

[179] W. Tait. "Intensional interpretations of functionals of finite type I." J. of Symbolic Logic **32** (1967) 198–212.

[180] W. Tait. "A Realizability Interpretation of the Theory of Species." Logic Colloquium, Ed. R. Parikh, Springer Verlag Lecture Notes **453** (1975).

[181] M. Takahashi. "A proof of cut-elimination theorem in simple type theory." J. Math. Soc. Japan **19** (1967).

[182] G. Takeuti. "On a generalized logic calculus." Japan J. Math. **23** (1953).

[183] G. Takeuti. "Proof theory." Studies in Logic **81** Amsterdam (1975).

[184] R. E. Tarjan. "Efficiency of a good but non linear set union algorithm." JACM **22,2** (1975) 215–225.

[185] R. E. Tarjan, J. van Leeuwen. "Worst-case Analysis of Set Union Algorithms." JACM **31,2** (1985) 245–281.

[186] A. Tarski. "A lattice-theoretical fixpoint theorem and its applications." Pacific J. Math. **5** (1955) 285–309.

[187] D.A. Turner. "Miranda: A non-strict functional language with polymorphic types." In Functional Programming Languages and Computer Architecture, Ed. J. P. Jouannaud, Springer-Verlag LNCS 201 (1985) 1–16.

[188] R. de Vrijer "Big Trees in a $\lambda$-calculus with $\lambda$-expressions as types." Conference on $\lambda$-calculus and Computer Science Theory, Rome, Springer-Verlag LNCS **37** (1975) 252–271.

[189] D. Warren "Applied Logic - Its use and implementation as a programming tool." Ph.D. Thesis, University of Edinburgh (1977).